(2)

AD-A256 006

REPORT DOCUMENTATIC

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Aug 1992	Final 1Oct89-31Dec91

4. TITLE AND SUBTITLE
Query Optimization and Planning in Object-Oriented Knowledge Bases (91)

5. FUNDING NUMBERS
AFOSR-89-0004
90
61102F
2304/A2

6. AUTHOR(S)
Phillip Chn-Yu Sheu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Center for Computer Aids for Industrial Productivity
Rutgers University

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 92 0 8

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFSOR/NM
BL.Dg 410
Bolling AFB DC 20332-6448

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFOSR-89-0004
90

DTIC
ELECTE
OCT 7 1992
S C D

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution unlimited.

12b. DISTRIBUTION CODE
91L

13. ABSTRACT (Maximum 200 words)

The purpose of the project entitled "Query Planning and Optimization in Object- oriented Knowledge Bases" sponsored by AFOSR-90-0004 is to extend a deductive object base with knowledge-based problem solving and planning, which is intended to realize the concept of very-high level programming in a database system. The input to such a system is a specification of the problem to be solved (as a set of goals) and the output is a solution of the problem, where the knowledge-based problem solving system deals with problems that do not change the state of a database and the planning system processes goals that require some state changes in the database.

In our approach, the knowledge-based problem solving system stores a set of problem models (such as graph problems) so that an input problem can be matched through an object-oriented specialization/generalization process. If no problem models can be matched by a given problem, the user should be provided with a high-level programming system that allows a top-down problem solving process be carried out until some matches can be found at detailed implementation stages. For the planning system, we have realized that most of the conventional approaches based on the formulation of operations-preconditions-postconditions have been proved to be inefficient. We have classified general planning problems into several classes so that each class can be solved individually and efficiently. With this approach, each class of planning problems can be constructed as a problem model and included in the general problem solving system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
N/A	39
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

11/19

92-26579

Final Technical Report Submitted to

THE AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

Project Title:

QUERY OPTIMIZATION AND PLANNING IN OBJECT-ORIENTED KNOWLEDGE BASES

Award Number:

90
AFOSR ~~89~~-0004

Principal Investigator

Dr. Phillip Chen-Yu Sheu
Dept. of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08855-0909
(908) 932-5388

QUERY OPTIMIZATION AND PLANNING IN OBJECT-ORIENTED KNOWLEDGE BASES

Dr. Phillip Chen-Yu Sheu
Department of Electrical and Computer Engineering
and
CAIP
Rutgers University
Piscataway, NJ 08855
(908)932-5388

Table of Contents

FINAL TECHNICAL REPORT

1. SUMMARY

1.1 PROJECT OBJECTIVES AND ACCOMPLISHMENTS

The purpose of the this project is to extend a deductive object base with knowledge-based problem solving and planning, which is intended to realize the concept of very-high level programming in a database system. The input to such a system is a specification of the problem to be solved (as a set of goals) and the output is a solution of the problem, where the knowledge-based problem solving system deals with problems that do not change the state of a database and the planning system processes goals that require some state changes in the database.

In our approach, the knowledge-based problem solving system stores a set of problem models (such as graph problems) so that an input problem can be matched through an object-oriented specialization/generalization process. If no problem models can be matched by a given problem, the user should be provided with a high-level programming system that allows a top-down problem solving process be carried out until some matches can be found at detailed implementation stages.

For the planning system, we have realized that most of the conventional approaches based on the formulation of operations-preconditions-postconditions have been proved to be inefficient. We have classified general planning problems into several classes so that each class can be solved individually and efficiently. With this approach, each class of planning problems can be constructed as a problem model and included in the general problem solving system.

Our accomplishments can be summarized as follows:

1. Within an object-oriented knowledge base framework, we have developed the necessary constructs to define problem models. Matching between a given problem and problem models is accomplished through theorem proving.
2. We have developed an object-oriented knowledge base programming environment in which object-oriented programs can be specified and executed easily and efficiently.
3. We have designed an object-oriented planning system that employs a high-level query language as the goal specification language. We have classified planning problems according to different constraints. With such, we can concentrate on classes of planning problems that allow efficient solutions. These problems include simple operator ordering problems, consumer ordering problems, producer ordering problems, and consumer-producer ordering problems.
4. We have found that most planning problems that require sequencing of operations that compete for space belong to the above classes of planning problems. Consequently, the previous approaches that universally employ the pre-condition/post-condition formulation is neither appropriate nor necessary, and efficient solutions can be developed with algorithmic approaches.

This report summarizes the theories and algorithms devised for items 1-3 listed above. Publications for item 4 are included in the appendix.

1.2 PUBLICATIONS PRODUCED

The following list summarizes the publications produced under this project:

[1] Yoo, S.B. and Sheu, P. C-Y., "Execution and Optimization of Database Programs in an Object-Oriented Symbolic Information Environment," (to appear) *IEEE Transactions on Data and Knowledge Engineering*, 1992.

[2] Lee, C.T., and Sheu, P. C-Y., "A Divide and Conquer Approach for Robot Path Planning," (to appear) *IEEE Transactions on System, Man, and Cybernetics*, 1992.

[3] Sheu, P. C-Y. and Xue, Q., *Intelligent Robotic Planning Systems*, World Scientific Publishing, 1992. (in press)

[4] Yoo, S.B. and Sheu, P. C-Y., "A Knowledge-based Program Transformation System," submitted to *IEEE Transactions on Data and Knowledge Engineering (revised)*.

[5] Sheu, P. C-Y., and Silver, D., "Extending Object-Oriented Databases with Geometric Problem Solving and Visualization," submitted to *Computer and Graphics*, 1991.

[6] Xue, Q., Maciejewski, A. and Sheu, P. C-Y., "Determining Collision-Free Joint Space Graph for Two Cooperating Robot Manipulators," submitted to *IEEE Transactions on Robotics and Automation*, 1990 (revised).

[7] Xue, Q., and Sheu, P. C-Y., "Path Planning for Reconfigurable Robots,", submitted to International Journal of Robotics and Manufacturing, 1990.

2. PROGRAM TRANSFORMATION

Programming activities are knowledge-intensive, where extensive knowledge of application domains and programming languages is required. Even though there have been numerous approaches in the field of program transformation and verification [PaSt83] [Feat86] [MiDG86], their practical utility is still limited. This is partially due to the inability to properly manage the large amount of knowledge from different domains. For this reason, the role of knowledge management for various problems in software engineering is getting more attention from researchers [Bars87]. Recently, several software engineering environments have been designed with strong supports from knowledge bases or data bases [Pene86] [Clem88] [Estu86] [HuKi88] [SmKW85].

This section describes a Knowledge-Based Program Transformation System (KBPTS) that has been designed on top of an object-oriented knowledge base for the purpose of automatic program transformation and optimization. In KBPTS, a program can be specified by means of a variation of C++ which allows object classes and their associated methods be functionally specified before implemented. In the object-oriented knowledge base, a collection of abstract algorithms are stored as a library of algorithms. Like application programs, the functionality of each abstract algorithm is given in additional to the implementation. A program in KBPTS is developed by first specifying its functionality. The transformation system then searches for an abstract algorithm whose functionality can match that of the program. If the search succeeds, the program is replaced by the implementation of the abstract algorithm (with proper instantiations), which is supposed to be efficient.

2.1 RELATED WORK

Work related to the programming system described in this report, can be classified into five categories: implementation of sets in object-oriented programming, object-oriented databases, program transformation, software reuse, and knowledge-based editors.

Implementation of Sets in Object-Oriented Languages

Among existing object-oriented programming languages, Smalltalk [GoAR89] may have the most object-oriented implementation of sets. Since *sets*, as well as *bags*, *arrays*, *dictionaries*, *sorted collections*, and others, are subclasses of the class *collection*, every instance of such classes is an independent object containing other objects. Depending on the class, various methods are available; some of which are methods to count, add, delete, copy, replace, or sort elements. For some classes, elements need not all be of the same type.

The influence of Smalltalk is evident in two languages [BGGH91]. The first, *Objective C*, is a superset of C. In addition to arrays and structures which are a part of C, it supports sets, dictionaries, and ordered collections which are based on the collection classes found in Smalltalk. The second language, *Actor*, also has collection classes which are taken directly from Smalltalk. Finally, there is one very popular object-oriented language, *C++* which, like *Objective C*, is a superset of C. There is no implementation of sets or any collection classes in *C++* other than the basic C arrays and structures. In [Koen92], several aspects of designing a container or collection class were considered in extending *C++*.

Despite that the provision of sets is an important feature of a declarative programming language, to our knowledge very few of object-oriented programming languages have listed declarative programming as its targeted goal; this is reflected by the fact that no further specification constructs such as variable quantifiers have been provided in such languages beyond sets.

Object-Oriented Databases

Several object-oriented databases have been proposed; a partial list includes GEMSTONE [MaSt86][CoMa84], Iris [Fish87], Ariel [Macg85], EXODUS [Care86], Trellis/Owl [Obri86] and POSTGRES [RoSt87] [StHH87]. Most of these systems have been designed to simulate semantic data models by including mechanisms such as abstract data types, procedural attributes, rules, inheritance, union type attributes and shared subobjects. Declarative programming in such systems is confined to be declarative retrieval of persistent objects.

Most recently, a number of object-oriented databases have been available commercially; some examples are Versant, O2, and ObjectStore [CACM91]. Instead of providing a query language, most of them require persistent objects be accessed directly from within an object-oriented program. Most of the above systems have been implemented with an extended language compiler and a separate database system so that accesses to persistent objects/data are implemented by an object storage system. To our knowledge, little consideration has been given to global optimization. On the other hand, although the lack of a query language can eliminate the gap between a database system and a programming language, it also causes declarative programming an even more remote goal of such systems.

Program Transformation

Program transformation includes predefined transformations (e.g., rewriting rules) and program constructions from a high level nonexecutable specifications to a low level executable form. Existing transformation systems can be divided into two classes: those that perform transformations automatically and those which are guided by users. The CIP project [CIP84] [BMPP89] focused on correctness-preserving and source-to-source program transformation at different levels of abstraction. The development process is guided by the programmer who has to choose appropriate transformation rules. The user guidance accomplishes the creative part in the development process. DEDALUS [MaWa79] and KBSA [PrSm88] attempted to automate the transformation selection process. DEDALUS was able to create a program, a correctness proof, and a proof of termination for programs of a limited scope. DEDALUS selects candidate rules by pattern-directed invocations and applies those rules sequentially. KBSA focused on automatic algorithm design, deductive inference, finite differencing, and data structure selection. Given a problem description, KBSA generates an optimal program through correctness-preserving transformations. One of the major problems with existing automatic program transformation systems is that most of them try to transform a program from scratch; consequently the lack of driving force of a design process can only lead to limited successes in practical applications. Even though some search approaches such as cost functions and efficient search methods have been employed, global strategies have yet been integrated effectively [Scha90].

Software Reuse

Software reuse appears in two levels of abstraction: reuse at the code level and reuse at the specification level [Dill88]. While code-level reuse involves modifying existing code [PrFr87], specification-level reuse is based on an external, often formal, program specification. Existing methodologies include program transformation [Chea84] [BoMu84] and software library [BABK87][WoSo88][NTFT91]. Program transformations are used to refine a specification or an abstract program defined in a very high level language into a program written in a target language [KaGa87]. Software libraries require the ability to locate the appropriate software components based on users' requests. Most of such systems employ certain kind of indexing techniques or syntactic matching for the purpose of searching; and a more flexible and efficient matching mechanism remains to be developed [Dill88].

Knowledge-Based Editors

Simple program editors have been extended to be more powerful ones. Some incorporate an understanding of the syntactic structure of the program being edited [MeFe81] [TeRe81]. This makes it possible to support operations based on the parse tree of a program (e.g., inserting, deleting, and moving between nodes in the parse tree). Syntax-based editors also ensure the syntactic correctness of the program being edited. KBEmacs [Wate85] extended program editors further by including an understanding of the algorithm structure of the program. By comparing the algorithm structures with programming cliches, which are standard models of solving programming problems, KBEmacs can intelligently assist programmers. KBEmacs assists programmers to construct programs more rapidly and more reliably by combining or modifying existing algorithmic cliches. The idea of using algorithmic cliches is similar to that of using abstract algorithms in KBPTS. One difference is that cliches are domain dependent reusable components while abstract algorithms are general ones which can be applied to problems in various domains.

2.2 DECLARATIVE OBJECT-ORIENTED PROGRAMMING

For the purpose of declarative programming, KBPT extends the C++ programming language with the constructs for functional specifications and associative programming, which are described in the following subsections.

2.2.1 FUNCTIONAL SPECIFICATIONS

IN KBPT, functional specifications are accomplished with the declaration of sets and the availability of quantifiers for logical expressions:

Set Classes

Given a class α, the class of all possible ordered sets which can be derived from instances of α is declared as:

```
class set_of_α {
  ...
  //methods
```

...
}

The following declaration defines a set *a* of class α:

set_of_α a;

Set Projection

Given a set or an object *a* of class α, the following notation designates the projection of *a* on attributes *A* 1, ..., *An*:

$a_{A1,...,An}$

Head and Tail

The function *head*() applying to a set returns the first element of the set; the function *tail*() returns the remaining of the set. The symbol *NIL* designates the empty set.

Universal Quantifier

A variable in a logical expression can be universally quantified by the quantifier:

(forall <variable_id> in <set_id>)

Existential Quantifier

A variable in a logical expression can be existentially quantified by the quantifier:

(exist <variable_id> in <set_id>)

Membership

The following function returns 1 if *<variable_id>* is an element of *<set_id>*:

<set_id>:member(<variable_id>);

□

Based on the above, a class declaration can define the functionality of each method and any logical property of its instances in additional to the structure of the class. The general form of a class declaration is:

```
class <class_id> {
     ...
     <class_id> [/<variable_id>] <method_id> (parameter_1,domain_1,....,
                                        parameter_n:domain_n);
      [<= logical expression;]
      ...
      [<= logical expression;]
} [<= logical expression]
 ....
 [<= logical expression]
```

In the above, each logical expression associated with a method, if specified, describes the desired relationships among the parameters, the target object (which is represented as *Self* [1]), and any returned object (which is represented as a variable that is defined after the returned class). Multiple logical expressions can be associated with a method; each of which designates an alternative functional specification. The symbol "+*Self*+" designates the updated value of *Self* in case the method changes the contents of the object. On the other hand, each logical expression associated with the class, if specified, describes the logical property of each instance. Multiple logical expressions can be associated with a class; each of which designates an alternative description of the class. Such descriptions are typically used to define derived classes and can be used in the program transformation process. A set class cannot be associated with any logical expression (but its methods can) as its properties are defined by the class it is derived from.

Example 2-1

The following declarations define the classes for an airline reservation system.

```
class city {
     public:
       //attributes and methods
       ...
};

class set_of_city {...};

class flight {
     public:
       city source, destination;
       float fare;
       void addfare(int amount);
          <= (+Self+ = Self + amount)
       //others
```

1 Note that *Self* and *self* are different; the latter is a pointer in C++.

```
    ...
};

class set_of_flight { ... };

class airline {
    public:
      set_of_city cs;
      set_of_flight fs;
      ...
      int connection(city s,city t,set_of_flight c,float fare);
         <= Self.cs.member(s) &&
              Self.cs.member(t) &&
              (c.head().source == s) &&
              Self:connection(c.tail().destination,t,fare1)) &&
              (fare = c.head().fare + fare1))
         <= Self.cs.member(s) &&
              Self.cs.member(t) &&
              (c.head().source == s) && (c.head().destination == t) &&
              (c.tail() == NIL) && (fare == c.head().fare)

      set_of_flight /d cheapest_connection(city s,city t,float fare);
         <= (connection(s,t,d,fare) == 1) &&
              !((exist c in Self.fs) (connection(s,t,c,fare1) &&
              (fare1 < fare))

       //others
      ....
}
```

□

Example 2-2

The following declarations define a class *father* and a derived class *ancestor*:

```
class father {
    char father[30], child[30];
    //others
    ...
};

class ancestor {
    char ancestor[30], descendant[30];
    //others
```

```
    ...
}; <= (exist s in father) (exist u in ancestor) ((s.father == Self.ancestor) &&
      (s.child == u.ancestor) && (u.descendant == Self.descendant))
   <= (exist s in father) ((s.father == Self.ancestor) &&
      (s.child == Self.descendant)
```

□

2.2.2 ASSOCIATIVE PROGRAMMING

The availability of sets as described in the last subsection allows objects be retrieved in an associative fashion. The following functions/statements are used to access the elements in a set:

1. *<set_id>:insert(<variable_id>);*
2. *<set_id>:delete(<variable_id>);*
3. *(foreach <variable_id> in <set_id>) statement;*

Example 2-3

Assume the following declarations:

```
class rectangle {
    public:
        vertex a,b,c,d;
        int intersect(rectangle r);  //test if two rectangles intersect
        int size(); //returns the size of a rectangle
        void plot(); //plot a rectangle
        //others
        ....
};

class vertex {
    public:
        float x,y;
        //others
        ...
};

class block {
    //attributes
    void plot(); //plot a block
    ....
```

```
};

class on_top {
    public:
        block top,bottom;
        //others
        ...
};

class set_of_block {...};
class set_of_on_top {...};
class set_of_rectangle {...};

set_of_block sb;
set_of_on_top sot;
set_of_rectangle sr;
block b;
op_:op a;
rectangle s,t,u;
```

The following are some example statements which access objects associatively:

```
//plot pairs of rectangles of sr which intersect each other
(foreach t in sr)
  (for each u in sr)
     if (t.intersect(u) == 1) {t.plot(); u.plot();}

//plot the smallest rectangle in sr
(foreach t in sr)
  if !((exist s in sr) (s.size() > t.size()) t.plot();

//plot each block which does not support any other block
(foreach b in sb)
  if !((exist a in sot) (a.bottom == b)) b.plot();
```

□

2.3 OBJECT-ORIENTED LOGIC SYSTEM

A set of class definitions as described in Section 3 can be translated into an object-oriented logic system. Formally, we define an object-oriented logic system to be a two-level system. The first level, or the *object level*, is a tuple $L_O = (O, G_O, D_O)$, where O is a first order object language, G_O is an object representation of O, and DL_O is a set of deductive laws. Similarly, the second level, or the *schema level*, is a tuple $L_S = (S, G_S, D_S)$, where S is a first order object language, G_S is an object representation of S, and DL_S is a set of deductive laws.

Consider an object base G_O, namely a set of classes and their associated methods. We define the first order schema language consisting of: a set of constants (which are started with an upper case letter), a set of variables (written in upper case letters), and the following predicates (in lower case) to describe object classes and relations:

1. *class* $(a, a_1, \ldots, a_n)$ is true if a is the name of a class of objects, and the attributes of each object of class a is $a_1, \ldots, a_n$. The symbol *set_of_a* designates the class of all possible ordered sets which can be derived from the objects in class a.
2. a :*method* $(m, d_1, \ldots, d_n)$ is true if a is a class, m is a method, and the domain of the i^{th} parameter of the method is d_i.
3. *attribute(a,b,c)* is true if the attribute b of class a has the the domain c, where c is a set.
4. The predicate *instance_of(a,b)* is true if object a is an instance of class b; the predicate *member_of(a,b)* is true if object a is an instance of set b.

Similarly, we define the first order object language consisting of: a set of constants (written in lower case letters), a set of variables (written in upper case letters), an n-place predicate symbol m for each of the n-ary method m (For simplicity, we shall assumed that all method names are distinct) and the following predicates (in lower case) to describe objects and relationship among a set of objects:

1. The predicate a:$m(x_1, \ldots, x_r)$ is true if the method m of some class is applied to the object a of the same class with the arguments $x_1, \ldots, x_r$ of legal values.
2. The predicate *a(t)* is true if t is an instance of class a. The predicate *set_of_a(t)* is true if t is an instance of the class *set_of_a* (i.e., t is a set whose elements are of class a).
3. The predicate *member_of(a,b)* is true if the object a is an element of the set b. The notation $[H|T]$, where H and T are variables or constants, designates a set whose first element is H and the rest of the set is T.

Finally, following the syntax and semantics of PROLOG, both a schema-level deductive law and an object-level deductive law are expressed in the form of:

f :- $f_1, f_2, \ldots, f_n$.

In both cases, f is called a *derived predicate*. If a perdicate f is defined in terms of:

f :- e_1.

...

f :- e_m.

its semantics is

f <=> $e_1 \mid e_2 \mid \ldots e_m$.

If $m = 1$, this shall result in f <=> e_1.

For simplicity, from now on we shall use the notation

$class(a,a_1:d_1,....,a_n:d_n)$

in place of the set of predicates:

$class(a,a_1,....,a_n)$
$attribute(a, a_1,d_1)$
...
$attribute(a, a_n,d_n)$

Example 2-4

Suppose we have an object class called *city* with only one attribute, *state*, whose domain is *string*, and an object class called *flight* with the following attributes:

1. *source*, whose domain is *city*;
2. *destination*, whose domain is *city*;
3. *fare*, whose domain is *float*;

Also assume that we have a class called *airline* with the following attributes:

1. *cs*, whose domain is *set_of_city*;
2. *fs*, whose domain is *set_of_flight*;

Associated with the class *airline*, assume we have a method call *connection* which takes two cities as the input and returns a set of flights which connect the two cities. The structure of this system can be described as follows, where expressions at schema level and expressions at object level are separated by a line. The same convention will be followed in the remaining of the report.

```
class(city,state:string)
class(flight,source:city,destination:city,fare:float)
class(airline,cs:set_of_city,fs:set_of_flight)
airline:method(connection,set_of_flight,city,city,float)
airline:method(cheapest_fare,city,city,float)
airline(A) :- instance_of(A.cs,set_of_city), instance_of(A.fs,set_of_flight).
---------------------------------------------------------------------------
A:connection(C,S,T,Fare) :-
   member_of(S,A.cs), member_of(T,A.cs),
   member_of(F,A.fs), (F.source = S), (F.destination = T),
   (C = [F]), (Fare = F.fare).

A:connection(C,S,T,Fare) :-
   member_of(S,A.cs), member_of(T,A.cs),
   member_of(F,A.fs), (F.source = S),
   A:connection(C1,F.destination,T,Fare1),
   C = [F|C1], (Fare = F.fare + Fare1).
```

A:cheapest_connection(D,S,T,Fare) :-
A:connection(D,S,T,Fare),
~(A:connection(C,S,T,Fare1), (Fare1 < Fare))

□

Similar to the two-level structure described above, given a class declaration presented in the general form, we say a logical expression is presented at object level if it is associated with a method and we say a logical expression is presented at schema level if it is associated with a class. The class declaration can be translated into an object-oriented logic system according to the following rules:

Quantifiers

A logical expression which is presented at object level and is existentially qualified in the form of *(exist t in c) d* is translated to *c(t), d* (if *c* is a class) or *member_of(t,c), d* (if *c* is a set). A logical expression which is presented at object level and is universally qualified in the form of *(forall t in c) d* is translated to *~(c(t), ~d)* (if *c* is a class) or *~(member_of(t,c),~d)* (if *c* is a set).

For the schema level, the same rules as described above apply except any expression of the form $\alpha(a)$ in the translated expression is replaced by *instance_of*(a,α) and any expression of the form *set_of*_$\alpha(a)$ is replaced by *instance_of*$(a,$*set_of*_$\alpha)$.

Structures

A class definition presented in the form of

class a {
domain_1 attribute_1;
...
domain_1 attribute_1;
//methods
} <= c_1
...
<= c_n

is translated to the following at the schema level:

class(a,attribute_1,...,attribute_n)[2]
attribute(a,attribute_1,domain_1)
...
attribute(a,attribute_n,domain_n)
a(Self) :- δ, c_1;

---

2 If an attribute is presented in upper case, it should be converted to lower case.

a(Self) :- δ, *c_n;*

where δ = *instance_of(attribute_1,domain_1)*, ..., *instance_of(attribute_n,domain_n)*. In case no *c_i* is specified, it is translated to:

class(a,attribute_1,...,attribute_n)
attribute(a,attribute_1,domain_1)
...
attribute(a,attribute_n,domain_n)
a(Self) :- δ.

Set Classes

A set class definition presented in the form of

class set_of_a {
  //methods
  ....
}

is translated to the following at schema level:

class(set_of_a)

Methods

A method definition presented in the form of

d/s c:m(parameter_1:domain_1,...,parameter_n:domain_n)
 <= *c_1*
 ...
 <= *c_n*

is translated to:

c:method(m,d,domain_1,...,domain_n)[3]

Self:m(s,parameter_1,...,parameter_n) :- δ', *c_1.*
Self:m(s,parameter_1,...,parameter_n) :- δ', *c_n.*

[3] If an attribute is presented in upper case, it should be converted to lower case. The variable *Self* can be replaced by any variable which is distinct from the others. In this case all instances of *Self* in *c_1*, ..., *c_n* should be replaced.

where δ' = *domain_1(parameter_1),...,domain_n(parameter_n)*. Note that the convention used in this report is that after translation, the first parameter of a method predicate corresponds to *Self*, and the second parameter corresponds to the returned object (if specified).

Sets

If both functions *head* and *tail* appear in a logical expression and are applied to the same object a of class *set_of_*α in the form of *a.head*() and *a.tail*() at object level, all instances of *a.head*() are replaced by a variable T and all instances of *a.tail*() are replaced by a variable S, where S and T are two variables which are distinct from the others. In the mean time, the following expression should be added to the translated expression:

$$\alpha(T),\ set\_of\_\alpha(S),\ A = [T|S]$$

If only *head* is applied to an object a of class *set_of_*α at object level, all instances of *a.head*() are translated to a variable T, where T is a variable which is distinct from the others. In the mean time, the predicate $\alpha(T)$ should be added to the translated expression. On the other hand, if only *tail* is applied to an object a of class *set_of_*α at object level, all instances of *a.tail*() are translated to a variable S, where S is a variable which is distinct from the others. In the mean time, the predicate *set_of_*$\alpha(S)$ should be added to the translated expression.

For the schema level, the same rules as described above apply except any expression of the form $\alpha(a)$ in the translated expression is replaced by *instance_of*(a,α) and any expression of the form *set_of_*$\alpha(a)$ is replaced by *instance_of*$(a,$*set_of_*$\alpha)$.

Membership

A function call of the form *a:member(b)* at both levels is translated to *member_of(b,a)*.

□

Example 2-5

With the above rules, the declarations made in Example 2-1 can be translated to the declarations presented in Example 2-4.

□

In addition, the expression δ' will be omitted for the examples for clarity.

2.4 OBJECT UNIFICATION

The presence of variables and constants at object level which are structured objects makes unification at that level a rather complicated task. At the first glance, given a predicate $c(A)$ and assuming the structure of c has been declared as $class(c,a_1:d_1,....a_n:d_n)$, it can be translated to the following set of predicates:

$c(A)$
$attribute\_value(A,a_1,A_1)$
...
$attribute\_value(A,a_n,A_n)$

where a predicate $attribute\_value(a,b,c)$ is true if object a has c to be the value of its attribute b. Now, any object expressed as $A.a_j$, $1 \le j \le n$, can be translated to A_j. The same rule can be applied recursively if any of the A_j's is a structured object. This mechanism seems to work well if the type of A is known exactly. However, if c is *object* (which means a can essentially be any type of object) or some unknown attribute of A is referenced (in the form of $A.B$, for example, where B is a variable), the above mechanism would be stucked. In the following, we shall extend the conventional unification algorithm in order to handle structured objects in general. Before proceeding, let us recall that the *disagreement set* of a nonempty set W of expressions is obtained by "locating the first symbol (counting from left) at which not all the expressions in W have exactly the same symbol, and then extracting from each expression in W the subexpression that begins with the symbol occupying at that position" [ChLe69].

The unification algorithm is extended to include structured objects as follows:

Object-Oriented Unification Algorithm

step 0
Retrieve the types of each expression if known.

step 1
$k = 0$, $W_k = W$, $\alpha_k = \phi$, $\beta = \phi$;

step 2
If W_k is a singleton, stop with success and return α_k. Otherwise, find the disagreement set D_k of W_k.

step 3
If there exist elements u and v in D_k, consider the following:

1. If both u and v are predicate symbols, u and v cannot be unified (as they are different) and stop with failure.
2. If $u = A_1.A_2...A_n$ and $v = B_1.B_2...B_m$, where each A_i, $1 \le i \le n$, or B_j, $1 \le j \le m$, is a constant or a variable:
   2-a If u and v cannot be of the same type with a unifier or with a unifier which has not been applied before and backtracking is possible, backtrack to the previous decision point;

otherwise stop with failure.

2-b If u and v can be of the same type with a unifier δ' which has not been applied before, add this step as a decision point. Let $\delta = \{(u \bullet \delta')/u, (v \bullet \delta')/v\}$. Also let $\beta = \beta \cup \{(\delta' \bullet u)/u, (\delta' \bullet v)/v\}$. If at this point there exists a set of unifiers of the form $\{w_1/y_1,....,w_r/y_r\}$, where each w_i is of the form $D_1.D_2...D_q.T_i$ for which T_i is a constant or a variable and each y_i is of the form $C_1.C_2...C_p.S_i$ for which S_i is a constant or a variable, consider the following. If $\{T_1,....,T_r\}$ covers all the attributes of $D_1....D_q$ and $\{S_1,....,S_r\}$ covers all the attributes of $C_1...C_p$, then add $D_1...D_q/C_1...C_p$ to δ. If $\{S_1,....,S_r\}$ covers all the attributes of $C_1....C_p$ but $\{T_1,....,T_r\}$ does not cover all the attributes of $D_1....D_q$, then add $D_{T1,...,Tr}/C_1...C_p$ to δ. Otherwise go to step 4.

step 4

Let $\alpha_{k+1} = \alpha_k \bullet \delta$, $W_{k+1} = W_k \bullet \delta$.

step 5

$k = k + 1$ and go to step 2.

□

Example 2-6

Consider the following two expressions, assuming *airline*(G) and *airline*(A), where the class *airline* is defined as in Example 2-1 and translated as in Example 2-4:

$$W = \{p(G.ES, G.NS, G), p(A.fs, A.cs, A)\}$$

According to the extended unification algorithm, initially β is ϕ. The unifier $\delta' = \{A/G, fs/ES\}$ unifies $G.ES$ and $A.fs$. Let $\delta = \{(G.ES \bullet \delta')/E.GS, (A.fs \bullet \delta')/A.fs\} = \{A.fs/G.ES, A.fs/A.fs\}$. Also set $\beta = \beta \cup \{A.fs/G.ES, A.fs/A.fs\} = \{A.fs/G.ES, A.fs/A.fs\}$. At the end of the first iteration, $W_1 = \{p(A.fs, G.NS, G), p(A.fs, A.cs, A)\}$. Similarly, a unifier for the second argument can be obtained as $\{A.cs/G.NS, A.cs/A.cs\}$ and β becomes $\{A.fs/G.ES, A.fs/A.fs, A.cs/G.NS, A.cs/A.cs\}$. At this point $A.fs$ and $A.cs$ cover all the attributes of A and $\{G.ES, G.NS\}$ covers all the attributes of G (based on their types), so that the unifier $\{A/G\}$ is added, and the resulting set of unifiers is returned successfully.

□

2.5 DEDUCTIVE UNIFICATION

A simple solution to the problem of mapping abstract algorithms to application algorithms can be proceeded as follows:

1. Construct abstract object classes and their associated methods in the object level.
2. Compare the application with the abstract classes and their associated methods; if a match can be identified, those abstract algorithms whose functionalities can be matched are instantiated.

We consider a library of algorithms as a collection of useful methods. Such algorithms should be as abstract as possible so that they can be instantiated by most applications. As an example, we can define the abstract class *w_graph* (weighted_graph) and some methods which implement efficient graph-based algorithms as follows[4]:

Graph-Based Classes and Methods

```
class(node)
class(edge,v1:node,v2:node,w:float)
class(w_graph,ns:set_of_node,es:set_of_edge)
w_graph:method(w_path,set_of_edge,node,node,float)
-----------------------------------------------------------------
G:w_path(P:set_of_edge,A:node,B:node,W:float) :-
   member_of(E,G.es), (E.v1 = A), (E.v2 = B), (P = [E]), (W = E.w).

G:w_path(P:set_of_edge,A:node,B:node,W:float) :-
   member_of(E,G.es), (E.v1 = A),
   G:w_path(P1,E.v2,B,W1), (W = E.w + W1), (P = [E|P1])

G:shortest_path(P:set_of_edge,A:node,B:node,W:float) :-
   G:w_path(P,A,B,W),
   ~(G:w_path(P1,A,B,W1), (W1 < W))
```

If we compare the functionality of the method *shortest_path* and the functionality of the method *cheapest_connection*, we can find the following terms which syntactically correspond to each other:

vertex(V)	*city(C)*
edge(E)	*flight(F)*
w_graph(G)	*airline(A)*
G:path(P,A,B,W)	*A:connection(C,S,T,F)*
G:shortest_path(P,A,B,W)	*A:cheapest_connection(C,S,T,F)*

In conventional unification algorithms, two predicates which have different predicate heads cannot be unified. However, we know that the *shortest_path* algorithm in class *w_graph* can be used for finding the cheapest connection in the class *airline* by properly instantiating the variables in the *shortest_path* algorithm with those in the airline reservation system with the following substitutions:

---

[4] In the remaining of the report, for clarity, we shall use the notation *c:m(p1:d1,...,pn:dn)* in place of *c:m(p1,...,pn)* when the functionality of the method is given.

Variables: {A/G,A.cs/G.ns,A.fs/G.es,S/A,T/B,F/E,C/P,Fare/W,F.fare/E.w}
Predicates: {airline/w_graph,connection_graph/w_path,cheapest_connection/shortest_path}

This is an example of matching with analogy [Carb81][Ders86][NTFT91], and in order to perform this we need an *analogical* unification process. This can be accomplished by extending the object unification algorithm to include predicate symbols for unification. However, this approach is clearly purely syntactic. To unify two programs analogically, it is required that each program be described with the same number of predicates and for each predicate, with the same number of arguments. The predicates used in the application need to be carefully designed so that they can be syntactically unified by those associated with the abstract algorithms. Consequently, the user has to memorize a large number of predicates, and their semantics, in order to reuse the abstract algorithms.

Another problem associated with the above approach is that the analogical unification algorithm only considers the number of arguments when two predicates are matched; as a consequence some random substitutions may be produced. As an example, consider two predicates *is_equal_set*(S_1,S_2) and *is_equal_tuple*(T_1,T_2), where the former is true if two sets S_1 and S_2 are equal, and the latter is true if two tuples T_1 and T_2 are equal. According to the analogical unification algorithm, they can be unified. However, since the argument domains for the two predicates are different, the algorithm testing for the equality of sets should be fundamentally different from that for tuples.

To solve the problems with second-order unification, a little more thought suggests that the matching between an application program and an abstract program should be done in first order; this implies we should parameterize the structure of an abstract class and present it as a derived class. The concept of parameterization is similar to that of templates in C++ [Stro91]. However, in order to instantiate a template in C++, the programmer has to be aware of its existence. Declaring it as a derived class can eliminate such a need, which is the theme of this research, so that the association between an application and a template can be established transparent to the programmer. With such, we can establish the following principle of matching between two methods P and Q:

If P <=> Q then P can be used to solve Q, and vice versa

Example 2-7

As an example, we can rewrite the graph-based methods as follows:

```
class(w_graph,NS:set_of_X,ES:set_of_Y)
graph(G) :-
   instance_of(G.NS,set_of_X), instance_of(G.ES,set_of_Y),
   attribute(Y,A,X), attribute(Y,B,X), attribute(Y,C,float).
w_graph:method(w_path,P:set_of_Y,S:X,T:X,W:float,L:int)
-------------------------------------------------------
G:w_path(P:set_of_Y,S:X,T:X,W:float,L:int) :-
  member_of(E,G.ES), (E.A = S), (E.B = T),
  (P = [E]), (W = E.C), (L = 1).

G:w_path(P:set_of_Y,S:X,T:X,W:float,L:int) :-
  member_of(E,G.ES), (E.A = S),
```

```
  G:w_path(P1,E.B,T,W1,L1), (P = [E|P1]),
  (W = E.C + W1), (L = L1 + 1).

G:shortest_path(P:set_of_Y,S:X,T:X,W:float,L:int) :-
  G:w_path(P,S,T,W,L),
  ~(G:w_path(P1,S,T,W1,L), (W1 < W))
```

To make the example more interesting, let us assume that a connection between two cities is restricted to consist of either one flight segment or two flight segments. In addition, we assume that the method *connection* is only interested in computing the fare for a connection between two cities. Note that with this the number of arguments associated with the predicates *w_path* and *connection* are different.

```
class(city,state:string)
class(flight,source:city,destination:city,fare:float)
class(airline,cs:set_of_city,fs:set_of_flight)
airline:method(connection,C:set_of_flight,S:city,T:city,Fare:float)
airline:cheapest_fare(S:city,T:city,Fare:float)
-----------------------------------------------------------------
A:connection(C,S,T,Fare) :-
  member_of(F,A.fs), (F.source = S), (F.destination = T),
  (C = [F]), (Fare = F.fare).

A:connection(C,S,T,Fare) :-
  member_of(F,A.fs), (F.source = S),
  member_of(H,A.fs), (H.source = F.destination),
  (H.destination = T), (Fare = F.fare + H.fare).

A:cheapest_fare(S,T,Fare) :-
  A:connection(C,S,T,Fare),
  ~(A:connection(C1,S,T,F), (F < Fare)).
```

We shall see that the method *shortest_path* can be used to solve the problem of *cheapest_fare*:

1. The object $A_{cs\,fs}$ forms a *w_graph* object. This can be proved by the following substitutions:

   A.fs/G.ES
   A.cs/G.NS
   city/X
   flight/Y
   source/A
   destination/B
   fare/C
   $A_{cs\,fs}$*/G*

   so that

airline(A) =>
instance_of(A.fs,set_of_flight),
instance_of(A.cs,set_of_city).

airline(A), attribute(flight,source,city),
attribute(flight,destination,city), attribute(flight,fare,float) =>
graph($A_{cs\,fs}$)

2. The methods *w_path* and *shortest_path* can be instantiated according to the above instantiations:

$A_{cs\,fs}$:w_path(P:set_of_flight,S:city,T:city,W:float,L:int) :-
member_of(E,A.fs), (E.source = S), (E.destination = T),
(P = [E]), (W = E.fare), (L = 1).

$A_{cs\,fs}$:w_path(P:set_of_flight,S:city,T:city,W:float,L:int) :-
member_of(E,A.fs), (E.source = S),
A:w_path(P1,E.destination,T,W1,L1), (P = [E|P1]),
(W = E.fare + W1), (L = L1 + 1).

$A_{cs\,fs}$:shortest_path(P:set_of_flight,S:city,T:city,W:float,LL:int) :-
$A_{cs\,fs}$:w_path(P,S,T,W,L), (L <= LL)
~($A_{cs\,fs}$:w_path(P1,S,T,W1,L1), (L1 <= LL), (W1 < W))

3. It can then be proved that

A:connection(C,S,T,Fare) => $A_{cs\,fs}$:w_path(P,S,T,W,1)

with the following set of substitutions from the first law associated with *A:connection*:

S/S
T/T
F/E
C/P
Fare/W

It can also be proved that

A:connection(C,S,T,Fare) => $A_{cs\,fs}$:w_path(P,S,T,W,2)

with the following set of substitutions from the second law associated with *A:connection*:

S/S
T/T
F/E
[H]/P1

H.Fare/W1
Fare/W
1/L1
2/L

4. Similarly, it can be proved that

$A_{cs\,fs}$:*w_path(P,S,T,W,1)* => *A:connection(C,S,T,Fare)*

with the following set of substitutions in the first law associated with *A:connection*:

S/S
T/T
E/F
P/C
W/Fare

and prove that

$A_{cs\,fs}$:*w_path(P,S,T,W,2)* => *A:connection(C,S,T,Fare)*

with the following set of substitutions in the second law associated with *A:connection*:

S/S
T/T
E/F
P1/[H]
W1/H.Fare
W/Fare

4. Based on 2 and 3, we can conclude that

A:connection(C,S,T,Fare) <=> $A_{cs\,fs}$:*w_path(P,S,T,W,1)* | $A_{cs\,fs}$:*w_path(P,S,T,W,2)*

Subsequently, the following can be concluded:

$A_{cs\,fs}$:*shortest_path(P,S,T,W,2)* <=> *A:cheapest_fare(C,S,T,F)*

This is because

A:connection(C,S,T,Fare) <=> $A_{cs\,fs}$:*w_path(P,S,T,W,L)*, *(L <= 2)*

□

Example 2-8

As another example, consider the following two versions of *sort*:

Version 1:

```
set_of_integer:method(sort,B:set_of_integer)
-------------------------------------------------------
A:sort(B) :- A:permutation(B), B:sorted().
[]:sorted().
[H|T]:sorted() :- not (member_of(X,T), (X < H)), T:sorted().
```

Version 2:

```
set_of_flight:method(sort,B:set_of_flight)
set_of_integer:method(sorted',B:set_of_integer)
-------------------------------------------------------
A:sort(B) :- A:permutation(B), B_fare:sorted'(C).
[]:sorted'().
[H|T]:sorted'() :- [H|T]:sorted_1(0).
[H|T]:sorted_1(N) :- (N <= H), T:sorted_1(H).
```

It can be proved by induction that *[H|T]:sorted()* => *[H|T]:sorted_1(0)* and therefore *[H|T]:sorted()* => *[H|T]:sorted'(0)* as follows:

1. It is trivial that *[]:sorted()* => *[]:sorted(0)*.
2. Assume that *[H|T]:sorted()* => *[H|T]:sorted_1(0)* (the Hypothesis). Now the following can be proved:

   [H'|[H|T]]:sorted() => *not (member_of(X,[H|T]), (X <= H')), [H|T]:sorted()*.

   By the hypothesis and the above, and since *[H|T]:sorted_1(N)* => *(N <= H), T:sorted_1(H)*:

   ```
   [H'|[H|T]]:sorted()
      => (H <= H'), [H|T]:sorted().
      => (H <= H'), [H|T]:sorted_1(0).
      => (H <= H'), (0 < H), T:sorted_1(H).
      => (0 < H), (H <= H'), T:sorted_1(H).
      => (0 < H'), [H|T]:sorted_1(H').
      => (0 < H'), [H|T]:sorted_1(H').
      => [H'|[H|T]]:sorted_1(0).
   ```

Similarly, we can prove that *[H|T]:sorted'()* => *[H|T]:sorted(0)* and conclude that *[H|T]:sorted'()* <=> *[H|T]:sorted(0)*.

□

3. OPTIMIZATION OF QUERY PROGRAMS

We define an object-oriented database program to be a set of statements (function calls in a LISP-flavored programming language) whose execution is sequenced by a set of control constructs. These statements in general operate on a set of programming objects (i.e., variables and constants) and database objects which are classified into different types. The major difference between a programming object and a database object is that the latter is persistent. However, the contents of a programming object can be assigned to a (compatible) database object and vice versa. From performance point of view, we feel that a major difference between a database programming system and an ordinary programming system should be that a database program needs to be evaluated by the database programming system but not by some extension of an ordinary compiler due to the large volume of data involved. On the other hand, it would be inappropriate to loosely couple an ordinary compiler (with a pre-processor) and a database query optimizer due to the communication overhead and the lack of global considerations.

This section studies the approach to globally optimizing the evaluation of database programs within a prototyped object-oriented database programming environment OASIS (an Object-oriented And Symbolic Information System), which is a database system intended to extend the conventional UNIX programming environment with persistent customized objects, object-oriented database programming, and symbolic information management. The OASIS query languages extend conventional database query languages with procedural methods and general control statements. As the complexity of the languages makes it difficult, if not impossible, to devise a "query optimizer" based on a universally applicable algorithm, the OASIS query interpreter optimizes the performance of OASIS programs based on a collection of "basic patterns" for which each pattern is associated with a separate query optimization algorithm. Consequently, an OASIS program can be divided into a set of segments and each segment is optimized separately.

In this section, we describe the optimization techniques for a set of basic patterns consisting of iterative statements and a set of nested statements. Such statements occur most frequently in query programs and are different from traditional nested queries (which are mainly used for the purpose of aggregation) in nature. The optimization techniques discussed in the following include an extended decomposition algorithm, evaluation of multiple conditions, data dependence analysis, and optimization of queries with arbitrary nesting. The conventional query decomposition algorithm [WoYo76] is extended to incorporate the evaluation of procedural methods. When a series of conditional statements is included in a nested loop, these statements can be transformed into independent statements so that common subexpressions can be shared to reduce the evaluation cost. When update operations are included in nested statements, data dependencies among statements are taken into account for proper optimization. Finally, for a general query which is an arbitrary combination of basic patterns, a global optimization strategy is discussed.

3.1 PREVIOUS WORK

In the past, several database programming languages have been proposed and/or implemented [AtBu87]. Some database query languages can be embedded in a host programming (e.g., SQL in PL/I [Astr76] and QUEL in C [SWKH76]). To our knowledge, most of the above systems have been implemented with an extended language compiler and a separate database system so that accesses to persistent objects/data are optimized by the database system at the query level. Little consideration has been given to global program optimization as an ordinary compiler does.

Along another direction, extensive research has been reported on the subject of *multiple-query optimization* and evaluation of nested queries for relational databases. In general, multiple-query optimization procedures consist of two parts [PaSe88]: identifying common sub-expressions and constructing a global access plan. Although detection of common sub-expressions or applicability of access paths may be computationally intractable or even undecidable if a set of arbitrary sub-expressions is considered [Jark84], various approaches have been proposed [JaKo84] [GrMi81] [Jark84] [ChMi82] [ChMi86]. Given the information of sharing, several search heuristic algorithms have been discussed [GrMi80] [Sell88] [PaSe88] [PaTL89]. On the other hand, optimization of nested queries in a relational database such as SQL has been discussed extensively in [Kim82] [Kim84] and [GaWo87], for which the major concern for nested queries has been the treatment of aggregation functions.

Multiple queries and nested transactions, in general, can be regarded as special cases of database programs. Consequently, the techniques developed in these two areas can be applied to optimize qualified segments in a database program as we will describe later.

3.2 OVERVIEW OF AN OBJECT-ORIENTED AND SYMBOLIC INFORMATION SYSTEM

In this section, we briefly review the essence of OASIS and introduce the schema definition for a small database as an example.

3.2.1 THE ARCHITECTURE OF OASIS

The overall architecture of an OASIS environment is shown in Figure 3.1, which consists of a database, a knowledge base, a meta-knowledge base and an OASIS database programming language interpreter:

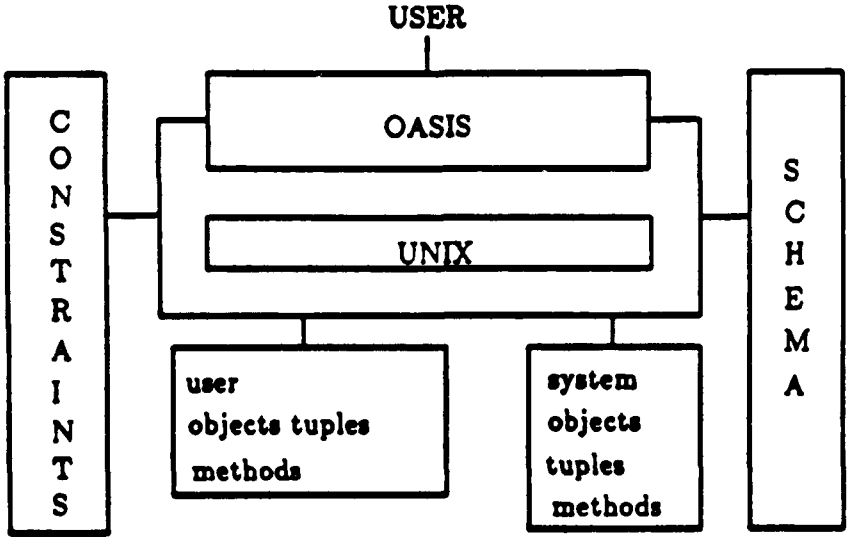


Figure 3.1. OASIS Architecture.

(1) The database contains a set of methods and a set of persistent objects that are organized into classes.

(2) On top of the objects and methods, there are a set of integrity constraints and a set of view definitions, stored in a textual form.

(3) An OASIS database program interacts with the OASIS environment with the OASIS interpreter. A user interacts with the OASIS system with functions written in OASIS-LISP (which is an extension of LISP) or OASIS-C (which is an interpretable version of C); both types of functions can be used in an interleaved way to accomplish a complex user query.

In OASIS, five classes have been predefined: integer, float, string, symbol[1] and list, where the first four are referred to as *primitive classes.* A *list* object is a composite object with three attributes: *name*, *car* and *cdr*, where *name* stands for the name of the list, *car* refers to the first element of the list, and *cdr* refers to a list that is composed of the remaining elements in the list. A summary of the database definition language of OASIS is given in Appendix A.

1. 3.2.2 A SMALL DATABASE AS AN EXAMPLE

In this section, we present a schema definition which describes a small world including various obstacles and moving cars. The meaning of each class and attribute is self-explanatory. Note that a class (recursively) inherits the attributes and the methods of its superclass unless the class overrides them. A class can redefine a method with the same name as in its superclass and rename an attribute in its superclass (e.g., the notation TID/OID can be used to change an attribute name OID into TID). For example, each entity in the class *triangle* has TID, SIZE, COLOR, and its three nodes as its attributes. The class hierarchy of this database is as follows:

```
<classes>
(defclass car_world (CWID int) (key CWID))

    <subclasses>
    (defsubclass obstacles car_world (OID/CWID int) (SIZE int) (COLOR string) (key OID))
        <subclasses>
        (defsubclass triangle obstacles (TID/OID int) (NODE1 position) (NODE2 position)
                    (NODE3 position) (key TID))
        (defsubclass rectangle obstacles (RID/OID int) (NODE1 position) (NODE2 position)
                    (key RID))
        (defsubclass circle obstacles (CID/OID int) (CENTER position) (RADIUS int)
                    (key CID))

    (defsubclass operator car_world (EMP_ID/CWID int) (DEPT string) (CAN_DRIVE car)
                    (key EMP_ID))
    (defsubclass car car_world (CAR_ID/CWID int) (YEAR int) (MODEL string)
                    (PERMIT int) (key CAR_ID))
```

[1] A symbol is a sequence of alpha-numeric characters.

(defsubclass station car_world (SID/CWID *int)* (PLACE *position)* (PERMIT *int)*
(key SID))

<methods>
; move an object *'a'* from *'b'* to *'c'*
(defmethod (move int) (a car_world) (b position) (c position) (...))

; rotate an object *'a'* by *'d'* degree
(defmethod (rotate int) (a car_world) (d int) (...))

; return the distance between two objects *'a'* and *'b'*
(defmethod (distance int) (a car_world) (b car_world) (...))

; return the length of the shortest path between two objects *'a'* and *'b'*
(defmethod (shortest_path int) (a car_world) (b car_world) (...))

; Given two geometric object *a* and *b*, check if they are intersected or not.
; If intersected, return the size of the intersected area; otherwise return 0.
(defmethod (intersect int) (a car_world) (b car_world) (...))

3.2.3 STORAGE STRUCTURES

In OASIS, objects in a class are stored in the form of a relation, where each tuple corresponds to an object instance in the class. Considering this, in the remaining of this section, we use the term "relation" and the term "class" interchangeably. An attribute value in OASIS can be a nested object and retrieval of each nested object is done directly through its key identifier (whose value is stored as the value of the attribute). For simplicity, we assume a uniform cost for referencing an attribute value.

3.3. A DATABASE PROGRAMMING LANGUAGE AND BASIC PATTERNS

As described earlier, an OASIS database program interacts with the OASIS environment with the OASIS interpreter. A user interacts with the OASIS system with functions written in OASIS-LISP (which is an extension of LISP) or OASIS-C (which is an interpretable version of C); both types of functions can be used in an interleaved way to accomplish a complex user query. In the rest of this section, we shall concentrate on OASIS_LISP. The term "query" will be used interchangeably with the term "query program".

Two outstanding features of OASIS-LISP which may affect the evaluation significantly are as follows:

a) Procedural methods are allowed in queries.

b) Control structures are allowed in a program.

In the following, we briefly explain these two features:

METHODS IN QUERIES

Methods are customized procedures associated with object classes. Including methods usually saves additional programming effort by calling methods within a query program. For instance, the following query retrieves a list of triangles which intersect with a rectangle and the areas of the two objects are the same.

```
(forall v1 in triangle
    (forall v2 in rectangle
        (cond (and (ieq v1.AREA v2.AREA)
                   (intersect v1 v2))
              (retrieve v1.TID))))
```

Without the method *intersect* in the above example, we may need an additional program segment checking the intersection of two rectangles.

CONTROL STRUCTURES Op8

In OASIS-LISP, various control structures such as conditional statements and iteration loops are included to enhance the scope of traditional query languages; these include *while*, *do* and *forall*, where a *while* or *do* statement iterates until a condition fails or the induction variable reaches a preset limit and a *forall* statement iterates for each instance of a given set. One example usage of iterations is to realize a transitive closure. One can find all the connections between two nodes using the transitive closure of connections.

In OASIS-LISP, most interesting relational (or set-oriented) operations can be programmed using *forall* and *cond* statements. Consequently, the basic patterns of statements can be classified according to the *forall* and *cond* statements in a query.

3.3.1 CANONICAL *FORALL-COND* STATEMENTS

A canonical query consists of a set of successively nested *forall* statements and a *cond* statement in the inner most loop. It is in the following form:

```
(forall ... in R1
 ...
  (forall ... in Rk
    (cond (F action))...)
```

When the innermost *action* is a *retrieve* statement, this is equivalent to a relational query as follows (written in QUEL):

```
RANGE OF v1 IS R1
:
RANGE OF vi IS Ri
RETRIEVE
     WHERE condition
```

Because only arithmetic comparisons and aggregation methods are supported in a relational database, including procedural methods causes some problems to traditional relational query optimization strategies. In optimizing a query program, a canonical query can be considered as a basic pattern.

When a query program is processed, it should be transformed into a canonical query if possible. For example, consider the following query which includes a *cond* statement in the middle of a set of successively nested *forall* statements.

(forall v_1 *in* R_1
  (forall v_2 *in* R_2
    (cond F_1
      (forall v_3 *in* R_3
        (cond (F_2 *action))))))*

Applying the commutative law between selections and cross products, the above query can be transformed into a canonical query as follows:

(forall v_1 *in* R_1
  (forall v_2 *in* R_2
    (forall v_3 *in* R_3
      (cond ((and F_1 F_2*) action)))))*

3.3.2 NESTED STATEMENTS IN SUCCESSIVE *FORALL* STATEMENTS

Various statements can be nested in one or more successively nested *forall* statements. The basic patterns of nested queries can be classified as follows:

(a) A GENERAL *COND* STATEMENT (TYPE-GENERAL_COND)

If we extend a canonical query with a general *cond* statement, we can obtain a query of the form:

(forall ... in R_1
...
  (forall ... in R_k
    (cond (F_1 *action*$_1$*)*
      ...
      (F_n *action*$_n$*)...)*

where the generalized *cond* statement should be interpreted as:

(IF F_1 THEN DO *action*$_1$)
(ELSE IF F_2 THEN DO *action*$_2$)
...
(ELSE IF F_n THEN DO *action*$_n$)

(b) MULTIPLE *COND* STATEMENTS (TYPE-MULTIPLE_COND)

If multiple *cond* statements are included inside of a set of successively nested *forall* statements, we would obtain a query of the form:

(forall ... in R_1
...
  (forall ... in R_k

(cond (F_1 $action_1$))

...

(cond (F_n $action_n$)...)

Semantically, the *cond* statements in the above query should be processed sequentially for each instance of variable bindings (i.e., each tuple in the cross products of relations R_1, ..., R_k).

(c) NESTED *FORALL* STATEMENTS (TYPE-NESTED_FORALL)

When a *forall* statement is present with other statements (e.g., other *forall* or *cond* statements) at the same level of nesting, it cannot be included in the outer *forall* statements. For example, consider the following query

```
(forall ... in R1
 ...
  (forall ... in Rk
    (forall ... in R
      (retrieve ...))
    (cond (F action)...)
```

In this example, (*forall ... in R (retrieve ...)*) is an individual *forall* statement rather than a part of a successively nested *forall* statements. Typical nested queries in a relational language can be considered as instances of this type.

(d) ASSORTED STATEMENTS (TYPE-ASSORTED)

In general, an arbitrary combination of the statements available in OASIS-LISP can be nested. Besides *forall*, *cond*, and *retrieve*, a statement in OASIS-LISP could be a method which may be a data manipulation statement such as *append*, *delete* and *replace*. In this situation, optimization of a nested statement may be affected by the presence of other statements. In particular, when data manipulation statements are present along with other statements (e.g., *cond* and nested *forall*) inside of a set of successively nested *forall* statements, data dependences should be analyzed for proper optimization. As an example, consider the following query:

```
(forall ... in R1
 ...
  (forall ... in Rk
    (cond (F1 action1))
    (replace Ri.A (plus Ri.A 10))
    (cond (F2 action2))))
```

Because the attribute value R_i.A, where $1 \leq i \leq k$, are updated after the first *cond* statement in each iteration, F_2 should be evaluated according to the updated values.

In fact, a canonical query or a nested statement of type GENERAL_COND, MULTIPLE_COND or NESTED_FORALL is a special case of a type-ASSORTED statement. Given a type-ASSORTED

statement, those special cases should be considered first.

3.3.3 GENERAL MULTI-LEVEL STATEMENTS (TYPE-GENERAL)

We define a level of nesting to be set of a successively nested *forall* statements and the body of the innermost *forall* statement, where the body of the innermost *forall* statement can include another level of nesting recursively. Generally, a nested query may consist of multiple levels of nesting, where each level of nesting would be one of the basic patterns defined above (i.e, canonical queries and nested statements of types GENERAL_COND, MULTIPLE_COND, NESTED_FORALL, and ASSORTED).

3.4. PROCESSING A CANONICAL QUERY

In the previous section, we showed a canonical query is semantically equivalent to a relational query. However, including procedural methods introduces problems to conventional relational query optimization techniques. For a large database, an optimal nested-loop algorithm could be inefficient when the number of variables involved in a query is large. Realizing this, a non-linear search approach based on query decomposition is taken in OASIS. The definitions of the query decomposition algorithm [WoYo76] and connection graphs are summarized in Appendix C. The main idea of the decomposition algorithm can be summarized as follows:

(a) Perform lower-cost operations first, i.e., in the order of selection, equi-join, general-joins and Cartesian product.

(b) Keep the temporary relations small by selecting small relations first and disconnecting the graph if possible.

In order to evaluate a canonical query written in OASIS-LISP, the original query decomposition algorithm should be modified because procedural methods were not considered. While conditions in the original query decomposition algorithm can be considered as logical methods in OASIS (see Appendix A), a conjunct in OASIS-LISP can be a method of any type. In the following, we discuss how to decrease the number of method calls when the query decomposition algorithm is followed.

In a connection graph, a procedural method can be denoted by a rectangular node, where its input arguments and output arguments are represented by input arcs and output arcs, respectively. As an example, an extended connection graph for the following query is shown in Figure 3.2.

```
(forall v1 in car
   (forall v2 in station
      (forall v3 in operator
         (forall v4 in manager
            (cond (and (ieq v1.PERMIT v2.PERMIT)
                       (and (seq v1.MODEL v3.CAN_DRIVE)
                       (and (seq v3.DEPT v4.DEPT)
                       (and (shortest_path v1 v2 PATH)
                       (ile PATH 100)))))
                  (retrieve all))))))
```

In OASIS, procedural methods are procedures which can include either simple mappings or very expensive computations (e.g., matrix computations). If a procedural method has only one input

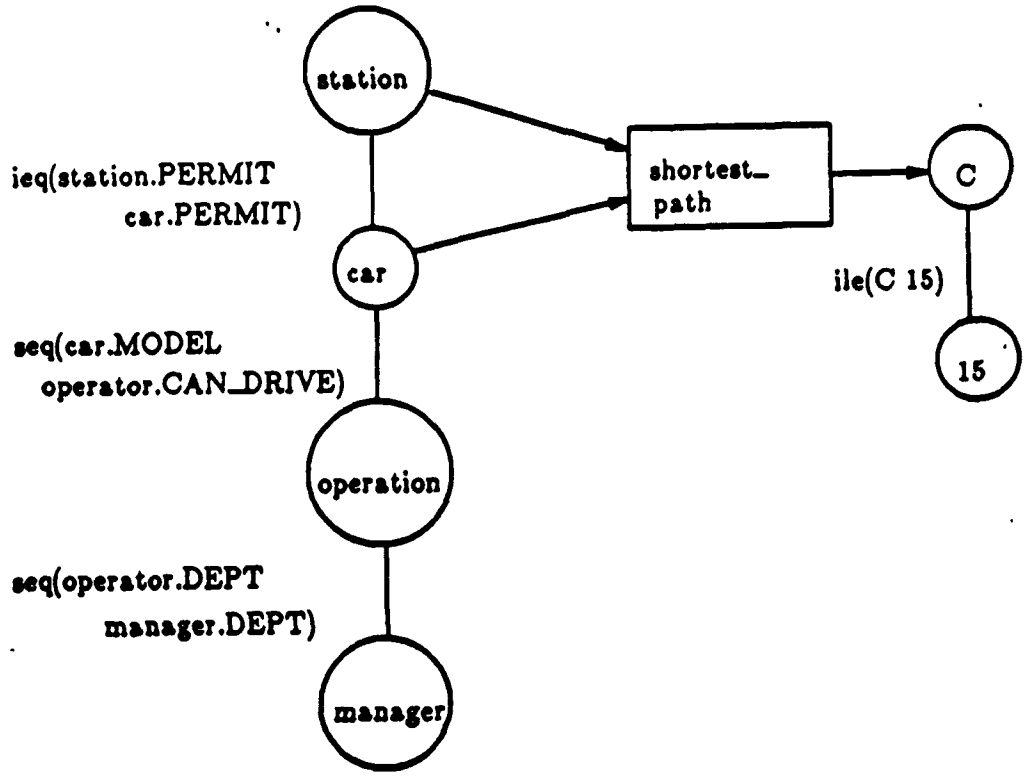


Figure 3.2. An extended connection graph.

argument and it is an attribute of a relation (we define this relation to be an *input relation* to the method), the method will be executed for each tuple in the relation. In this case, evaluation of the method does not change the number of tuples in the input relation. It attaches the values of output arguments to each tuple in the input relation. If the input values of a procedural method come from more than one relation and the relations are not connected by joins, the method should be evaluated for all possible combinations (of instantiations) for its input variables (i.e., all the tuples from the Cartesian product of the input relations).

Before a procedural method is evaluated, all the input relations of the method should have been instantiated or dissected because the values of input arguments are needed for evaluation. In the decomposition algorithm, evaluation of procedural methods should be included among dissections because instantiations can always be done without increasing the number of tuples in relations. During dissections, we can reduce the number of method calls by considering the effects of a dissection on an input node (i.e., a node which represents an input relation) or a node which is connected to an input node of the method.

When two relation nodes n and m are connected by a join edge, we define the reduction factor r_n^m associated with the edge to be

$$r_n^m = \frac{|m \ join \ n|}{|n|} .$$

Similarly, the reduction factor r_m^n can be defined as

$$r_m^n = \frac{|m \ join \ n|}{|m|} .$$

If an input node *in* for a procedural method is a candidate for the next dissection, we can compute the reduction factors for all the nodes $con_1, \cdots, con_k$ which are connected to *in*. If the value of any $r_{in}^{con_i}$, where $1 \le i \le k$, is less than one, dissecting con_j can reduce the number of method calls. Similarly, if a candidate node *cand* for the next dissection is connected to an input node *in* for a method to be evaluated, we compute the reduction factor r_{in}^{cand}. If r_{in}^{cand} is greater than one, the input node *in* should be dissected first to decrease the number of method calls. As an example, consider the connection graph shown in Figure 3.3, assuming the cardinalities of the involved relations are given as follows:

$|triangle| = 100$
$|rectangle| = 100$
$|triangle \ join \ rectangle| = 500$

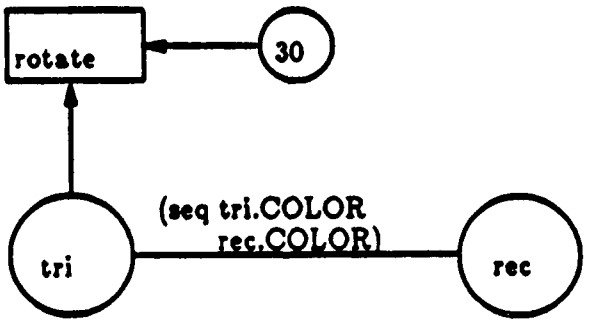


Figure 3.3. A connection graph including a procedural method.

According to the decomposition algorithm, ignoring the method *rotate*, either *rec* or *tri* can be selected for dissection. Assuming *rec* is selected, because the node *rec* is directly connected to the input node *tri* of the method *rotate*, we compute the reduction factor r_{tri}^{rec} before *rec* is dissected:

$$r_{tri}^{rec} = \frac{|triangle \ join \ rectangle|}{|triangle|} = 5$$

This implies that if the join is performed, the method *rotate* should be evaluated for each tuple in the result of the join, and the cardinality of which is five times as large as that of relation *triangle*. Considering this, the node *tri* should be dissected first, and, in this case, the number of method calls is 100.

If a method has more than one input relation, then any connection between the input relations should be considered for the method evaluation. When two input relations for a method are connected directly by an edge, evaluation of the edge may decrease the number of method calls because the method will be evaluated for only those pairs of tuples which satisfy the condition on the edge. This is a slight modification to the dissection operation because the method calls will be deferred until after the edge is evaluated. When two input relations for a method are connected indirectly (i.e., they are connected through some other nodes), we can compare the cardinalities of the resulting relation produced by the connections and the Cartesian product of the two input relations to choose a smaller one. If the former is chosen, evaluation of the method should be deferred until all the involved edges are evaluated.

Considering the above, the modified query decomposition algorithm can be summarized as follows:

ALGORITHM 3.1. Modified Query Decomposition Algorithm
INPUT: A connection graph for a canonical query.
OUTPUT: An access plan.

1) Do all instantiations. Here a method is instantiated (executed) if all of its input arguments have been instantiated.
2) Select a relation node n for dissection based on the original query decomposition algorithm.
   2.1) If n is an input relation for a method to be evaluated, compute the reduction factors with respect to all the nodes $con_1, \cdots, con_k$ connected to n. If any reduction factor $r_n^{con_i}$, where $1 \le i \le k$, is less than one, select the node con_j which produces the minimal $r_n^{con_j}$ for dissection.
   3.2) If the node n is connected to an input node in of a method to be evaluated, compute the reduction factor r_{in}^{n}. If r_{in}^{n} is greater than one, select the node in for the next dissection.
3) Perform dissection on the selected node n'. If at this point all the input nodes to a method are only connected through the method, evaluate the method for all the tuples in the Cartesian product of the input nodes.
4) Repeat steps 1-3 until no edge remains.
6) Return the Cartesian product of the relations saved so far.

□

Example 3-1. Consider the connection graph shown in Figure 3.2. According to the original decomposition algorithm, either *operator* or *car* can be selected for dissection. Assume $r_{car}^{operator}$ is less than one, where

$$r_{car}^{operator} = \frac{|car \ join \ operator|}{|car|}$$

According to step 2.1) of Algorithm 3.1, we dissect *operator* first. After the dissection, the relations *car* and *station* can be instantiated. Subsequently, the join between *operator* and *manager* is computed. Because the method *shortest_path* has two input nodes and they are connected directly, it should be evaluated for each tuple in the result of the join between the two relations *car* and *station*. Finally, all the *car-station* pairs with the shortest distance less than or equal to 15 are computed, and

the Cartesian product of such pairs and all the *operator-manager* pairs computed earlier is returned as the result.

□

The modifications introduced in Algorithm 3.1 may change the basic order of dissections in the original decomposition algorithm only when the change can reduce the number of method calls. In addition, when a node to be dissected is one of the input nodes of a method, evaluation of the method is deferred until all the input arguments are instantiated.

3.5 PROCESSING A TYPE-GENERAL_COND OR TYPE-MULTIPLE_COND QUERY

A type-GENERAL_COND or type-MULTIPLE_COND query may contain multiple conditional clauses (i.e., conditions and associated actions). Basically these conditional clauses can be evaluated sequentially by nested iterations. However, if the order of evaluation is not relevant to the meaning of the query (e.g., read-only queries in general), for each conditional clause, we can derive a relation whose tuples satisfy the condition using relational operations. Subsequently, evaluation of multiple conditions can be optimized by considering common subexpressions [Kim82] [Kim84]. A heuristic approach to optimize multiple conditions will be presented later by modifying the decomposition algorithm.

3.5.1 MULTIPLE CONDITIONS

We recite the general syntax of type-GENERAL_COND queries as follows:

(forall ... in R_1

...

(forall ... in R_k

(cond $(F_1$ *action*$_1)$

...

$(F_n$ *action*$_n)$...*)*

The simplest approach to processing a query of the above form would be to take a Cartesian product of all the relations involved in the *forall* statements, then test each condition in turn. Assuming that $|R_i| = n_i$ (for $1 \le k$), the cost of evaluating such a query would be of the order $O(n_1 \times ... \times n_k)$. However, if we collect only those tuples which will be used for the actions in the conditional clauses, performing the Cartesian product of all the involved relations can be avoided.

As mentioned earlier, the *cond* statement in the above is semantically equivalent to an IF-THEN-ELSE statement. Among the tuples in the Cartesian product of R_1, ..., R_k, only those which satisfy at least one of the conditions will be executed. In terms of relational algebra, one of the actions is executed for the following candidate relation:

$$R_{cand} = \sigma_F (R_1 \times ... \times R_k)$$

where $F = F_1 \vee ... \vee F_n$. The disjunction of conditions can be expanded using *unions*, i.e.,

$$R_{cand} = \sigma_{F_1}(R_1 \times ... \times R_k) \cup ... \cup \sigma_{F_n}(R_1 \times ... \times R_k)$$

Now the cost of evaluating the subqueries $\sigma_{F_i}(R_1 \times ... \times R_k)$, $1 \le i \le n$, can be decreased by sharing common subexpressions.

Similarly, consider a type-MULTIPLE_COND query in the following form:

(forall ... in R_1

...

(forall ... in R_k

(cond (F_1 *action*$_1$*))*

...

(cond (F_n *action*$_n$*)...)*

In this case, a candidate relation for each *cond* statement can be computed. The candidate relation R_{cand_i} for $action_i$, $1 \le i \le n$, contains only those tuples which will be actually used for the execution of $action_i$, $1 \le i \le n$:

$$R_{cand_i} = \sigma_{F_i}(R_1 \times ... \times R_k)$$

The evaluation cost can be reduced by considering the sharing of common expressions among R_{cand_i}, $1 \le i \le n$. In the next subsection, Algorithm 3.1 is further modified to evaluate multiple conditions.

3.5.2 PROCESSING MULTIPLE CONDITIONS

As discussed in [ChMi86], a connection graph can be extended as follows to represent multiple conditions. First, one candidate relation should be returned as the answer for each condition. Second, the priority in selecting the nodes should be changed because some instantiations may prevent later sharings.

Now, given a node for each relation R_i, $1 \le i \le n$, a conjunct in each condition can be added as an edge. The label of each edge is added with the condition number from which it comes. For example, considering the following query, its connection graph is shown in Figure 3.4.

C_1 : *(and (ile tri*.SIZE *10)*
(and (seq tri.COLOR *rec.COLOR)))*
C_2 : *(and (seq tri*.COLOR *rec.COLOR)*
(igt (distance rec cir) 50))

According to [ChMi86], in processing an extended connection graph, each condition is evaluated separately unless some sharing is possible. The basic two operations (i.e., instantiation and dissection) were modified as follows:

(a) After an instantiation has been performed on a relation, the edge, the node corresponding to the constant are deleted, and the relation node together are turned into a small node. If two conditions are identical, only one small node is created.

(b) When the join conditions are different between two nodes, one dissection will be done for each condition and a separate set of constant nodes is created. For two conditions that are identical, only one set of constant nodes is produced.

When the above two types of basic operations are executed, the execution cost can be reduced by sharing common subexpressions among conditions. However, in order to achieve the maximum

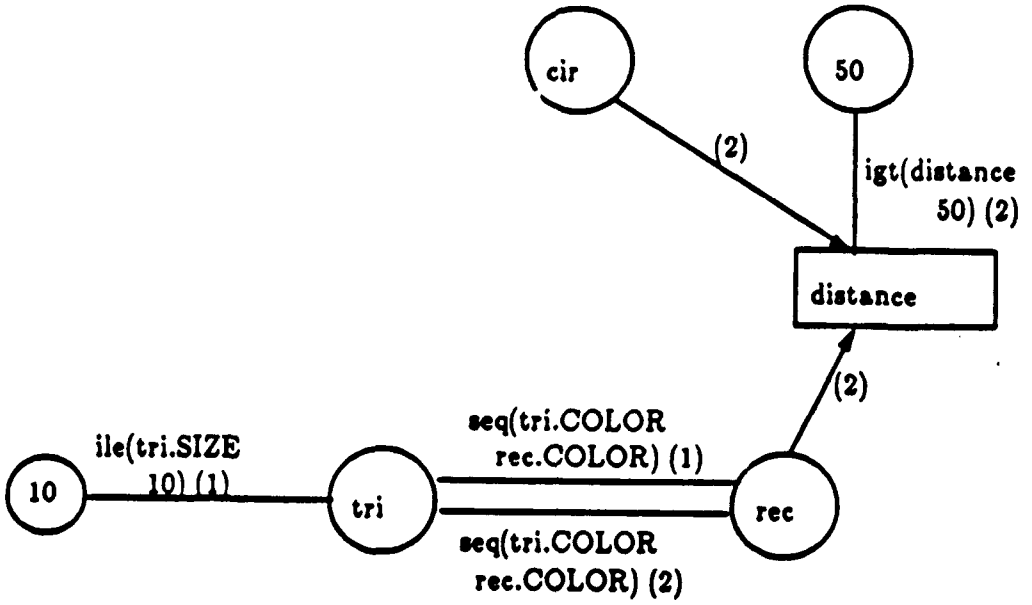


Figure 3.4. A connection graph for multiple conditions.

saving, a complete search with a combinatorial complexity is necessary [Jark84]. In most cases, a heuristic approach with a relatively small search space would be viable. In evaluating multiple conditions heuristically, the most important thing is to defer the instantiations properly. If two conditions share a join and one of these two includes a selection on one of the relations (see Figure 3.5 (a) for an example), obviously the common join should be executed first. If two conditions share a join and each of them has a different selection on one of the relations (see Figure 3.5 (b) for an example), the two relations have to be evaluated separately. However, as shown in Figure 3.5 (c), two selection conditions could be comparable (i.e., one condition subsumes the other). In other words, the result of one selection (*tri*.SIZE < 100) is a super set of the result of the other selection (*tri*.SIZE <10). In this situation, the more general selection (*tri*.SIZE < 100) can be executed first followed by the join operation. The other selection will be executed based on the result of the join.

When methods are considered, the order of evaluation with sharing of subexpressions as described above may conflict the procedure described in the previous section. In other words, evaluating common subexpressions may increase the number of method calls. To avoid this, we can dissect a input argument of a method if the cardinality of the input node is known to be increased (by computing the reduction factors) after including it in a common sub-expression. Considering the above, a modified decomposition algorithm that considers the evaluation of multiple conditions and methods can be summarized as follows:

ALGORITHM 3.2. Query Decomposition Algorithm for Multiple Conditions
INPUT: A connection graph for multiple conditions.

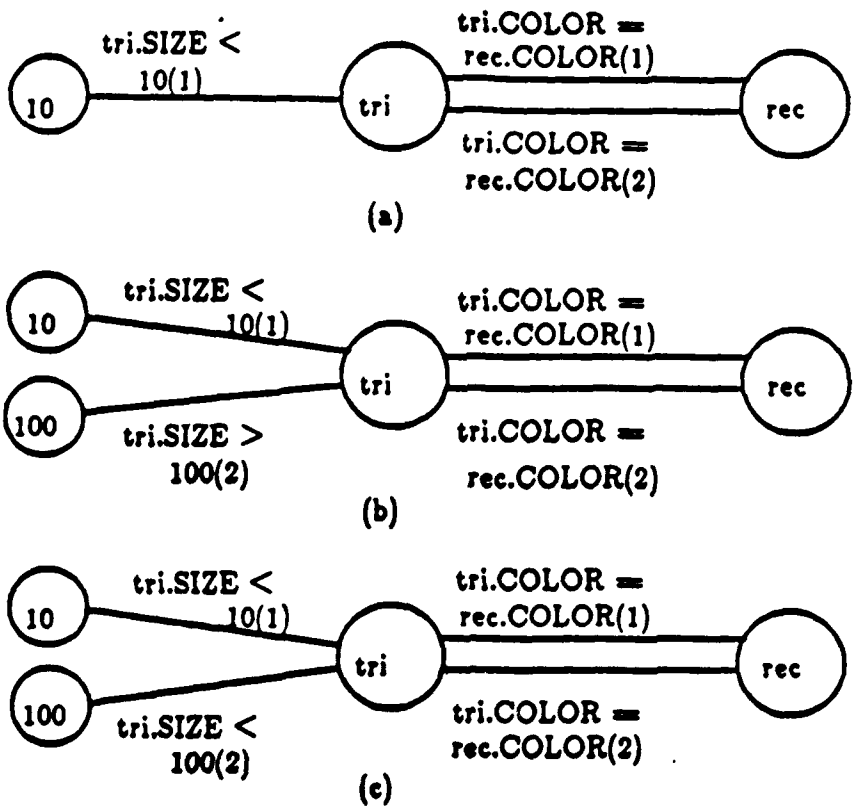


Figure 3.5. Examples of ordering common expressions.

OUTPUT: An access plan.

1) Do instantiations on relation nodes which are not incident upon any common edge. Here a method is instantiated only if all of its input relations have been instantiated.

2) Select a node n for dissection in the following order of preference:

   1.1) Select a node which is incident on a common join edge between two relation nodes for dissection. If the node is connected to an input node in of a method and the reduction factor r_{in}^{n} is greater than one, dissect in instead of n. If the node is connected to any constant with a selection edge, do the following:

      1.1.1) If the selection is common to the set of conditions which share the common join edge, execute the instantiation before the node is dissected.

      1.1.2) If the node is connected to a different constant for each of the conditions that share the common join edge and these selections are comparable, execute only the most general selection before the node is dissected. The other selections are done immediately following the dissection.

   1.2) Select a node which is incident on a common join edge between a relation node (i.e., itself) and a method for dissection. This choice can be superceded by the same considerations

listed in step 3 of Algorithm 3.1. If the node is connected to any constant with a selection edge, do the following:

1.2.1) If the selection is common to the set of conditions which share the common edge, execute the instantiation before the node is dissected.

1.2.2) If the node is connected to a different constant for each of the conditions that share the common edge and these selections are comparable, execute only the most general selection before the node is dissected. The other selections are done immediately following the dissection.

1.3) If no node can be selected in the above, select a node for dissection according steps 2 and 3 of Algorithm 3.1.

2) Do step 1 until no edge remains.

3) Return the Cartesian product of the relations saved so far for each condition.

□

Example 3-2. Consider the multiple queries shown in Figure 3.4. Initially, no instantiation is possible since none of them is shared. According to step 1.1 of Algorithm 3.2, the node *tri* is be selected for dissection in Step 1 (This is arbitrary, the node *rec* can be selected as well.) Assuming that the reduction factor between *tri* and *rec* is 5, *rec* is dissected instead of *tri*. The method *distance* with input node *rec* can be evaluated in Step 2. In Step 3, the common join with condition (*seq tri*.COLOR *rec*.COLOR) is performed. Subsequently, the selection (*ile tri*.SIZE 10) is be executed before the evaluation of the method *distance*.

□

The algorithm for multiple queries described in this subsection considers procedural methods in sharing of common subexpressions among multiple conditions. One minor modification made to the decomposition algorithm presented in [ChMi86] is the change of the order of preference in selecting operations. Clearly, this algorithm does not generate a globally optimal access plan while the methods in [Sell88] and [PaSe88] do. The procedure generated by this algorithm could be noticeably expensive compared with the case in which each common subexpression is extremely high while it can be avoided if each condition is processed separately. In order to prevent this situation, we could include some checking procedure prior to the evaluation of each common subexpression. For example, we can estimate the cardinality of the result relation for a common subexpression and take alternatives if it is too large.

3.5.3 PROGRAM TRANSFORMATION

Once each R_{cand_i}, $1 \le i \le n$, is obtained, a query of type-GENERAL_COND given earlier in this section can be transformed into the following:

$(forall\ v_1\ in\ R_{cand_1}$
$\quad action_1)$
$(forall\ v_2\ in\ (R_{cand_2} - R_{cand_1})$
$\quad action_2)$
$\vdots$
$(forall\ v_n\ in\ (R_{cand_n} - R_{cand_2} - \ldots - R_{cand_{n-1}})$

$action_n$)

Using a temporary relation, a number of subtraction operations can be saved in the above query:

(*forall* v_1 *in* R_{cand_1}
    $action_1$)
(*forall* v_2 *in* (R_{cand_2} - R_{cand_1})
    $action_2$)
(*set temp* ($R_{cand_1} \cup R_{cand_2}$))
(*forall* v_3 *in* (R_{cand_3} - *temp*)
    $action_3$)
(*set temp* (*temp* $\cup R_{cand_3}$))
⋮
(*set temp* (*temp* $\cup R_{cand_{n-1}}$))
(*forall* v_n *in* (R_{cand_n} - *temp*)
    $action_n$)

Similarly, a query of type-MULTIPLE_COND can be transformed into the following:

(*forall* v_1 *in* R_{cand_1}
    $action_1$)
(*forall* v_2 *in* R_{cand_2}
    $action_2$)
⋮
(*forall* v_n *in* R_{cand_n}
    $action_n$)

The evaluation cost of the query can thus be reduced by considering common-subexpressions in the transformed query.

3.6 PROCESSING A TYPE-NESTED_FORALL QUERY

When a *forall* statement or a set of successively nested *forall* statements is present with other statements in the body of another set of successively nested *forall* statements, a query is not canonical and a different optimization technique is needed. In OASIS, a type-NESTED_FORALL query is transformed to a canonical query if it is equivalent to a traditional nested query [Kim82]. If a type-NESTED_FORALL query cannot be transformed into a canonical one, the nested *forall* statement can be optimized by avoiding repeated processing of invariant computations inside of the statement. In this subsection, we describe an approach to detecting loop invariants inside of a nested *forall* statement. To start, given a nested *forall* statement within another nested *forall* statement, we define its associated *inside loops* to be all the *forall* statements included in the statement and its associated *outside loops* to be all the *forall* statements which iterate on the nested *forall* statement. As an

example, consider the following query:

```
(forall v1 in triangle
   (forall v2 in rectangle
      (forall v3 in circle
         (cond ((and (intersect v1 v2)
                     (seq v2.COLOR v3.COLOR))
                (retrieve v3.AREA into Temp)))
   (cond (ieq v1.AREA imax(Temp))
      (retrieve v1.TID))))))
```

We can derive the followings:

$$nested\ forall\ statement = (forall\ v_2\ in\ triangle\ \cdots\ )$$

$$inside\ loops = \{\ (forall\ v_2\ in\ rectangle\ \cdots\ ),\ (forall\ v_3\ in\ circle\ \cdots\ )\ \}$$

$$outside\ loops = \{\ (forall\ v_1\ in\ triangle\ \cdots\ )\ \}$$

Given a nested *forall* statement, a connection graph can be constructed for its associated inside loops. The connection graph for the nested *forall* statement (*forall* v_2 *in rectangle* ...) is shown in Figure 3.6. Note that as before, we use arrows to represent input arguments for procedural methods.

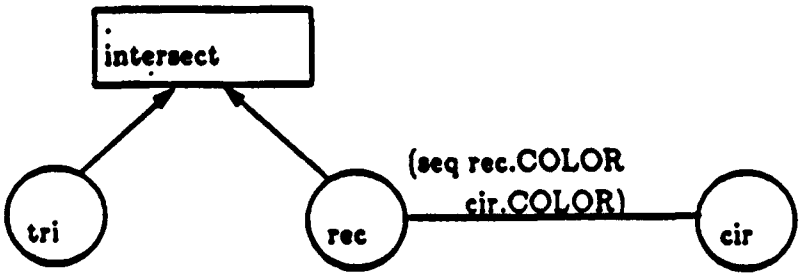


Figure 3.6. A connection graph.

Given a connection graph, we traverse it from all the node corresponding to the relations included in its outside loops (e.g., *triangle* in this example). In the traversal, we follow all incident arrows and edges on a current node, where arrows are traversed only in the forward direction. After the traversal, we can collect all the edges which have not been passed as loop invariants, and these edges can be precomputed outside of the nested *forall* statement only once. In this example, we start the traversal with the node *tri*. The traversal soon terminates at the node *intersect*; thus leave the edge between *rec* and *cir* not traversed. Consequently, the join between *tri* and *rec* is considered loop invariant and can be precomputed and saved. As a result, the above query can be transformed into

the following:

```
(forall v1 in triangle
   (forall v2 in new
      (cond (intersect v1 v2.RECTANGLE)
            (retrieve v2.AREA  into Temp)))
   (cond (ieq v1.AREA (imax Temp "" AREA))
         (retrieve v1.TID)))
```

where the temporary relation *new* can be computed as

```
(forall v3 in rectangle
   (forall v4 in circle
      (cond (seq v3.COLOR, v4.COLOR)
            (retrieve v3 and v4.AREA into new))))
```

3.7 PROCESSING A TYPE-ASSORTED QUERY

When a number of statements are arbitrarily combined and nested in a set of successive *forall* statements, evaluation or optimization of a nested statement may be affected by other nested statements. In this section, update statements such as *append*, *delete* and *replace* are considered.

3.7.1 EVALUATION OF ASSORTED STATEMENTS

A statement in OASIS could be a *forall* statement, a *cond* statement, a method, or an update statement. Even though a method can stand alone as a statement, in most cases methods are included in a *forall* or a *cond* statement as a part of its condition or its action.

Adjacent *forall* or *cond* statements can usually be evaluated with the consideration of common subexpression sharing unless update operations are included in one of these statements. If no update operation is included, connection graphs for these statements can be merged into one and Algorithm 3.2 can be applied to evaluate this graph. For example, the following type-ASSORTED query can be considered as two sub-queries, and the merged connection graph for this query is shown in Figure 3.7(a).

```
(forall v1 in triangle
   (forall v2 in rectangle
      (forall v3 in circle
         (cond ((and (seq v1.COLOR v2.COLOR)
                     (ieq v2.AREA v3.AREA)))
               (retrieve v3.AREA )))
      (cond (seq v1.COLOR v2.COLOR)
       (retrieve v1.TID))))
```

can be decomposed into two sub-qureies:

statement 1:

```
(forall v1 in triangle
   (forall v2 in rectangle
      (forall v3 in circle
         (cond ((and (seq v1.COLOR v2.COLOR)
                     (ieq v2.AREA v3.AREA)))
               (retrieve v3.AREA )))))
```

statement 2:

```
(forall v1 in triangle
   (forall v2 in rectangle
      (cond (seq v1.COLOR v2.COLOR)
       (retrieve v1.TID))))
```

The merged connection graph for this query is shown in Figure 3.7(a).

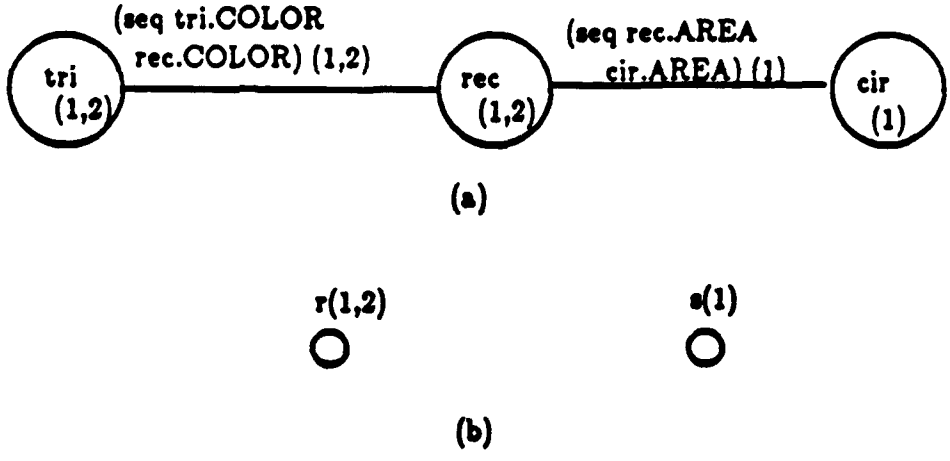


Figure 3.7. A merged connection graph for a type-ASSORTED query.

Note that in the connection graph shown in Figure 3.7(a), each relation node is marked with the statements in which it is iterated. In this example, relations *triangle* and *rectangle* are iterated in both statements, but relation *circle* is iterated only in statement 1. Given a merged connection graph, Algorithm 3.2 can be applied to process it until no edge remains. A candidate relation for each statement i can be generated by a Cartesian product of all the disjointed nodes marked with i.

The connection graph shown in Figure 3.7(a) can be processed by dissecting the node *rec* first since it disconnects the graph and it is incident on a common join edge. After this operation, two

nodes $r(1,2)$ and $s(1)$ remain as shown in Figure 3.7(b), where the node $r(1,2)$ represents the result of the join between the relation *triangle* and a tuple t in the relation *rectangle*, and (1,2) denotes that this node belongs to statements 1 and 2. Similarly, $s(1)$ stands for the results of the join between relation *circle* and a tuple t in *rectangle*, and it belongs to statement 1. As a result, the candidate relations for statements 1 and 2 can be generated as follows:

$$R_{cand_1} = \bigcup_{t \text{ in } r(1,2)} (r(1,2) \times s(1))$$

$$R_{cand_2} = \bigcup_{t \text{ in} r(1,2)} (r(1,2))$$

Evaluation of n consecutive statements by Algorithm 3.2 is viable when the summation of the cardinalities of all the candidate relations (i.e., $\sum_{i=1}^{n} |R_{cand_i}|$) is less than the cardinality of the Cartesian product of all the involved relations and there is a reasonable amount of sharing. If at least one statement runs on all the combinations of tuples (i.e., the Cartesian product) of the involved relations, applying Algorithm 3.2 cannot reduce the evaluation cost comparing to the simple nested iterative approach. A *cond* statement with a condition which is always true or a method which stands alone as a statement are examples. Given a sequence of statements nested in a set of successively nested *forall* statements, estimating $\sum_{i=1}^{n} |R_{cand_i}|$ and the amount of sharing prior to applying Algorithm 3.2 is necessary.

3.7.2 UPDATE OPERATIONS IN QUERY PROGRAMS

Evaluation of multiple statements is more restricted if update operations (i.e., *append*, *delete*, and *replace*) are involved. When update operations are included in a series of statements, some relations could be modified by a statement. Consequently, any statement which follows it and uses the modified relations should wait until the modification is done before it is evaluated. This is an example of *data dependence* [Kuck81], and we say that the latter statement *depends on* the former. Generally, when there are data dependences among nested statements, a statement cannot be evaluated separately using the candidate relations described in the previous subsection. For example, consider the following query:

```
(forall v1 in triangle
   (forall v2 in rectangle
      (forall v3 in circle
         (cond ((and (seq v1.COLOR v3.COLOR)
                     (ieq v2.AREA v3.AREA)))
               (replace v3.AREA (iadd v3.AREA 10))))
         (cond ((and (intersect v1 v3)
                     (ieq v2.AREA v3.AREA)))
               (retrieve (list v1.TID v2.RID v3.CID)))))
```

We have two statements nested (in parallel) in a set of nested *forall* statements:

```
S1 : (cond ((and (eq v1.COLOR v3.COLOR)
                 (ieq v2.AREA v3.AREA)))
```

(*replace* v_3.AREA (iadd v_3.AREA 10))))

S_2 : (cond ((and (*intersect* v_1, v_3)
(*ieq* v_2.AREA v_3.AREA)))
(*retrieve* (*list* v_1.TID v_2.RID v_3.CID)))

In the above example, S_1 updates the relation *circle*, and S_2 reads the value of the relation. The execution of S_2 should wait until S_1 is executed in each iteration. Because the relation *circle* might be updated again in the next iteration, the execution of S_2 cannot be postponed to the next iteration either. As a result, the nested statements S_1 and S_2 should be evaluated sequentially in each iteration of the *forall* loops. In the remaining of this subsection, we describe the data dependence among nested statements.

We define the *read_set* and the *write_set* for each statement as follows. The read_set of a statement is a set of relations whose contents are read by the statement. Similarly, the write_set of a statement is a set of relations whose contents are modified by the statement. This can be rewritten as

$$read\_set(S_i) \triangleq \{ R \mid \text{the contents of } R \text{ are read by } S_i \}$$

$$write\_set(S_i) \triangleq \{ R \mid \text{the contents of } R \text{ are modified by } S_i \}$$

Given a sequence of nested statements $S_1, \ldots, S_n$ in a set of nested *forall* loops, they can be numbered so that $i < j$ if S_i comes before S_j in the sequence. The procedure of finding the data dependence among a set of nested statements can be summarized as follows:

(1) Derive *read_set* and *write_set* for each statement S_i, $1 \le i \le n$.

(2) We have a dependence pair (S_i, S_j) if

i) $i < j$

ii) $write\_set(S_i) \cap read\_set(S_j) \neq \emptyset$

iii) $write\_set(S_i) \cap read\_set(S_j) \cap write\_set(S_k) = \emptyset$, for all $i < k < j$.

In each dependence pair (S_i, S_j), S_j depends on S_i.

Example 3-3. For the example query in this subsection, we can derive the following:

$read\_set(S_1) = \{ triangle, rectangle, circle \}$

$write\_set(S_1) = \{ circle \}$

$read\_set(S_2) = \{ triangle, rectangle, circle \}$

$write\_set(S_2) = \{ \}$

Through a dependence analysis, (S_1, S_2) is found to be a dependence pair because the set $\{ R \mid R \in write\_set(S_1) \cap read\_set(S_2) \}$ (which is $\{ circle \}$) is not empty.

□

Given a type-ASSORTED query which includes update operations, we search for any data dependence among nested statements as described above. If no data dependence is included in the query, it can be evaluated as described in the previous subsection. Otherwise, the query should be evaluated by nested iterations.

3.8 PROCESSING A TYPE-GENERAL QUERY

A type-GENERAL query may contain multiple levels of nesting, where each level of nesting would be one of the basic patterns, i.e., canonical, GENERAL_COND, MULTIPLE_COND, NESTED_FORALL, or ASSORTED. Given a type-GENERAL query, first it can be optimized globally by moving some computations to outer levels of nesting from inside. Then, it can be processed by applying the optimization techniques discussed earlier.

3.8.2 GLOBAL OPTIMIZATION

In a query with multi-levels of nesting, computation in a nested loop is supposed to be evaluated for each iteration of an outer loop. In this situation, if some computations in a nested loop could be executed in an outer loop without changing the results, the overall evaluation cost can be reduced.

Detection of loop-invariants has been discussed earlier in this section. Loop-invariants of a loop denote those computations whose results do not depend on the iteration variables of the loop. In some cases, even those computations which are not loop-invariants can be removed from a loop if necessary variables are saved properly. Generally, we can postpone any computation until its results are used by some others. The reason for postponing computations is that the amount of postponed computations may be reduced after they have gone through database operations such as joins and selections.

To optimize a query globally, levels are introduced in a connection graph for queries with multiple levels of nesting. Levels of nesting can be numbered by setting the outermost level to level 1 and increasing the number by one as it goes down to inner loops. Two adjacent levels are divided by a dotted line in a connection graph. A connection graph with the notation of levels for the following nested query is shown in Figure 3.8.

```
(forall v1 in obstacle
    (forall v2 in triangle
        (forall v3 in rectangle
            (cond ((and (setq INTSEC intersect(v2, v3) (setq S (get_area INTSEC)))
                (retrieve v3.AREA S into Temp1))
        (cond (ieq v2.AREA imax(Temp1 "" AREA)))
            (retrieve v2.TID Temp1.S into Temp2))
(cond (and (include Temp2.TID v1) ((igt Temp2.S 100)))
    (retrieve Temp2.TID))))))
```

In Figure 3.8, the dotted arrows designate direct copies between levels and they do not represent any computation. In this example, the derived attribute *S* in level 3 is not used until the outermost query (i.e., level 1) is evaluated. We can notice that the number of calls for the method *get_area* can be reduced significantly if we defer the evaluation until level 1. To avoid this, the input argument *INSEC* should be saved in temporary relations.

In order to generalize this observation, we need a simple data-flow analysis. Computations in a query program can be divided into two types: relational operations (e.g., join and selection) and procedural methods. For each operation C, we define the following:

USE[C] = the set of variables whose values are used by C.
GEN[C] = the set of variables whose values are generated by C.

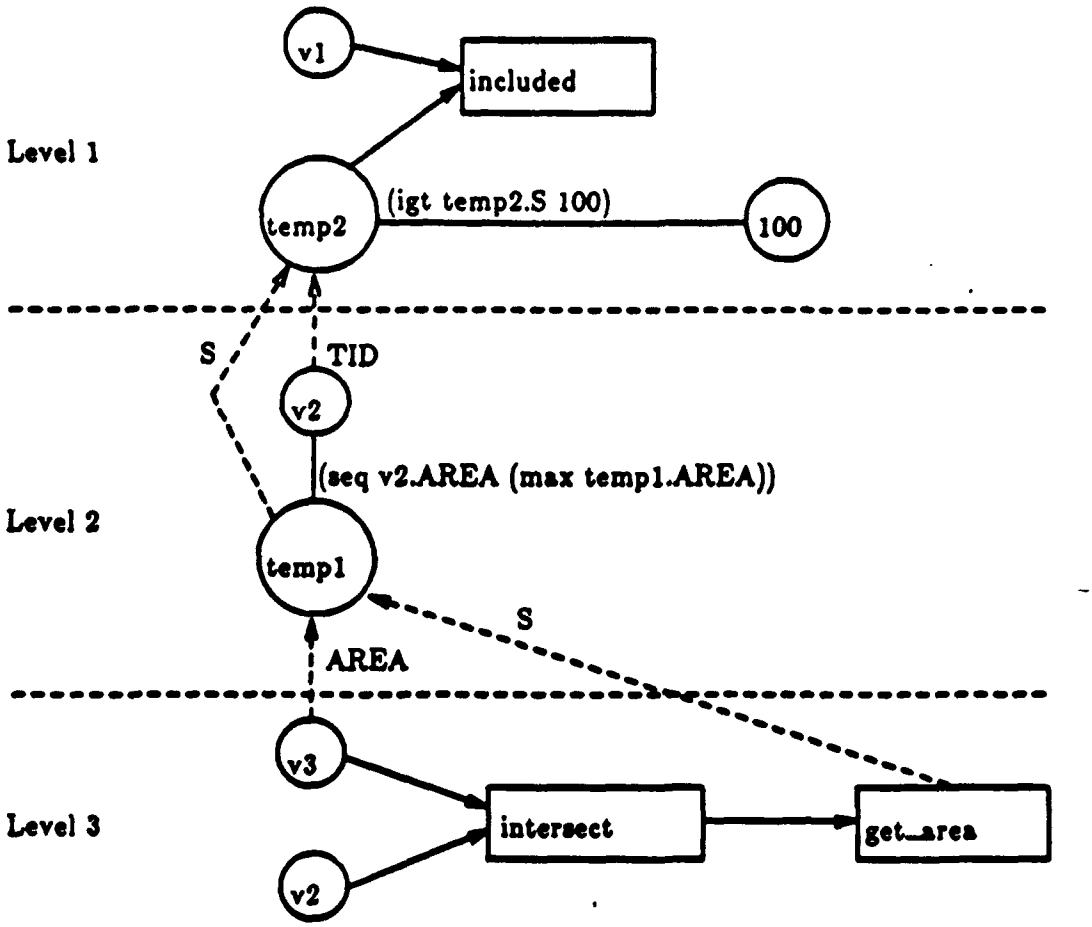


Figure 3.8. A connection graph for a nested query.

For a relational operation, the set *USE* includes all the relations and constants which are involved in the operation and the set *GEN* includes the result relation. For a procedural method, the set *USE* includes all the input arguments and the set *GEN* includes all the output arguments. For each level L_i, $GEN[L_i]$ includes the iteration variables which can be "used" in L_i and all the variables available from all the temporary relations in L_i. According to the above, $GEN[L_i]$ includes all the iteration variables for level L_j, where $1 \leq j \leq i$.

To analyze the data flows in each level L_i, we first derive the *USE* and *GEN* sets for all the operations included. For each *GEN* in L_i, search all *USE* sets in L_i to check if any variable in it is used in that level. If all the variables in $GEN[C]$ are not used, the evaluation of C can be moved to the next higher level, i.e., L_{i-1}. The details of this procedure are described in the next algorithm. Because it is very expensive to save all the input relations for relational operations, only procedural

methods are considered in the next algorithm.

ALGORITHM 3.3. Global Optimization
INPUT: A connection graph for a type-GENERAL query.
OUTPUT: An optimized type-GENERAL query.

Suppose the given query has n levels of nesting. For each level L_i, where i = from n to 1, do the following:

1) Derive $GEN[L_i]$ and GEN and USE for each operation in L_i.
2) For each procedural method M_j, search through all USEs to check any variable in it is used or not. If all the variables in $GEN[M_j]$ are not used at the level, do the following:
   2.1) If any variable in $USE[M_j]$ is an iteration variable, go to the beginning of Step 2 and consider M_{j+1}.
   2.2) Save all the variables in $USE[M_j]$ into a temporary relation and move M_j into the next level L_{i-1} with proper connections between members of $USE[M_j]$ and $GEN[C_j]$.
3) Repeat Step 2 until no method can be moved.

□

Example 3-4. Consider the connection graph shown in Figure 3.8. At level 3, we can derive the following:

$GEN[L_3] = \{ v_3, v_2 \}$
$USE[intersect] = \{ v_3, v_2 \}$
$GEN[intersect] = \{ INTSEC \}$
$USE[get\_area] = \{ INTSEC \}$
$GEN[get\_area] = \{ S \}$

With a simple search, we can find that the variable in $GEN[get\_area]$ (i.e., S) is not used by any other computations. Because $USE[get\_area]$ (i.e., *INTSEC*) is not an iteration variable (i.e., v_3 and v_2), the method *get_area* can be moved into level 2 by saving *INTSEC* into a temporary relation. Without *get_area*, the next iteration of Step 2 in Algorithm 3.3 will find that $GEN[intersect]$ (i.e., *INTSEC*) is not used, and therefore it can be moved up again. In this example, the method *intersect* cannot be moved because its *USE* includes some iteration variables.

After applying Algorithm 3.3 to levels 2 and 1, the connection graph is changed as shown in Figure 3.9 and the corresponding query program is as follows:

```
(forall v1 in obstacle
    (forall v2 in triangle
        (forall v3 in rectangle
            (cond ((setq INTSEC intersect v2 v3))
                (retrieve v3.AREA INTSEC into Temp1)))
        (cond (seq v2.AREA (imax Temp1 "" AREA)))
            (retrieve v2.TID Temp1.INTSEC into Temp2))
    (cond ((and (include Temp2.TID v1)
            (and ((setq S (get_area Temp2.INTSEC )
                (igt S 100))))
```

(*retrieve Temp2*.TID)))

□

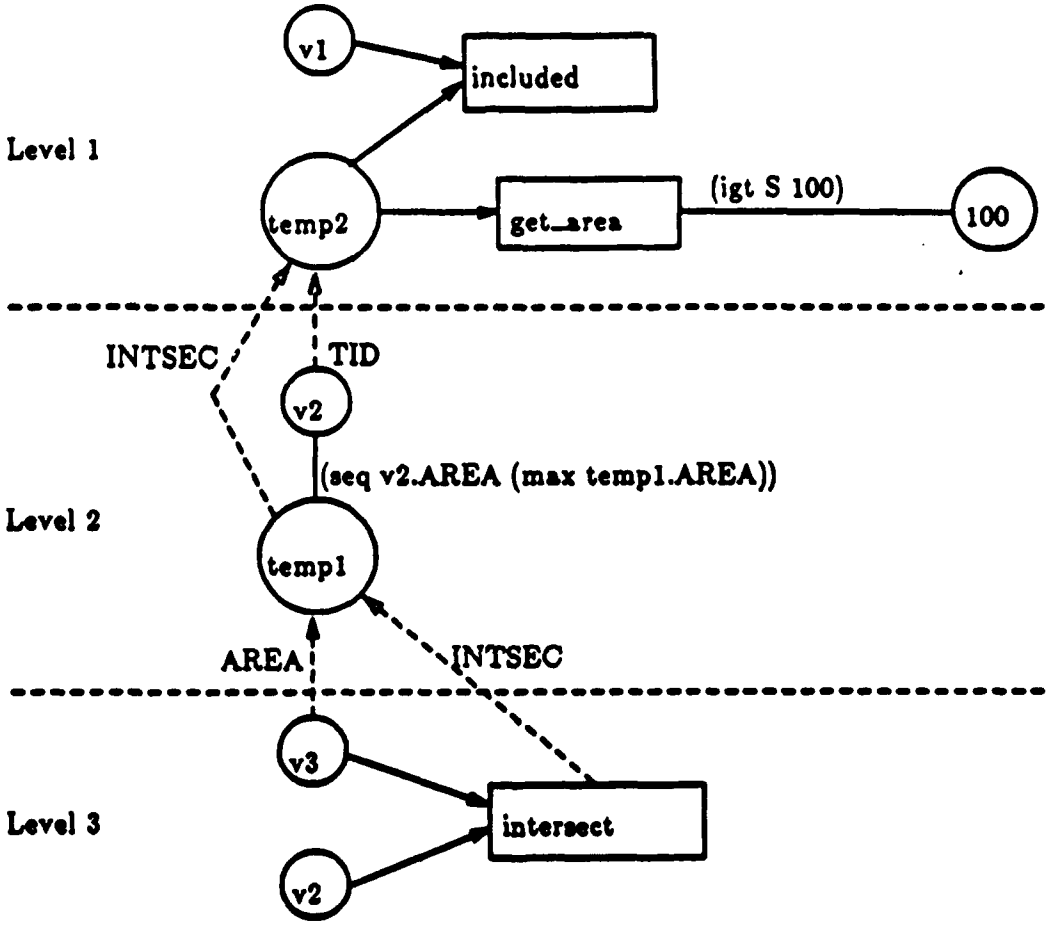


Figure 3.9. An optimized connection graph.

In the above example, the number of calls to the method *get_area* is reduced because the temporary relations have been operated by a set of selections before the method *get_area* is evaluated in the modified program. Once a type-GENERAL query is optimized by the above algorithm, each level of nesting can be optimized further by applying an appropriate technique discussed earlier. In the

next subsection, we describe the general procedure to evaluate a type-GENERAL query.

3.8.3 PROCEDURE OF EVALUATING A TYPE-GENERAL QUERY

Given a type-GENERAL query, it can be globally optimized first as discussed in the previous subsection. Further optimization and evaluation is based on the optimization techniques for basic patterns. One way to represent the structure of a given query is to use a query graph.

A query graph is a binary tree, where each node denotes a statement. Each node can have up to two children: a left child and a right child, where the left child represents the first statement among its nested statements and the right child represents the next statement in the same level of nesting. For example, the query graph for the transformed query in the previous subsection is shown in Figure 3.10.

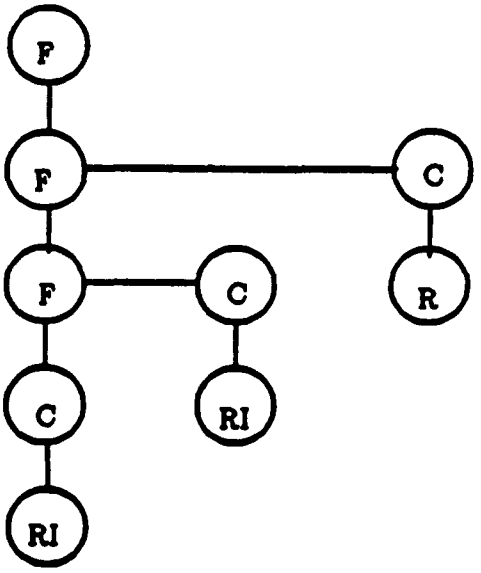


Figure 3.10. A query graph.

In this query graph, nodes are labelled by F, I, R and RI to represent statements of types *forall*, *cond*, *retrieve* and *retrieve_into*, respectively.

The optimization techniques discussed in this section can be divided into two types:

a) Type A:Techniques which transform a given query syntactically.
Techniques of this type are those developed for type-NESTED_FORALL queries (i.e., detecting loop-invariants and transforming them into canonical queries).

b) Type B:Techniques which speed up the evaluation of a given query.
Techniques of this type are those developed for canonical queries and for types GENERAL_COND, MULTIPLE_COND and ASSORTED.

Techniques of type A may change the structure of a given query or move some computations into upper levels. For this reason, techniques of type A should be applied prior to those of type B. Also, they should be applied in a bottom-up fashion (i.e., from the lower levels to the upper levels) so that any change in the structure of the query can be propagated to the upper levels.

Techniques of type B can include some modifications to the statements, but those modifications are internal and do not affect the overall structure of a given query. Techniques of type B are applied in a top-down fashion.

In summary, a general procedure for processing a type-GENERAL query is described in the following algorithm.

ALGORITHM 3.4. OPTIMIZATION OF A TYPE-GENERAL QUERY
INPUT: A query graph for A type-GENERAL query.
OUTPUT: An optimized evaluation procedure.

1. Apply Algorithm 3.3 to the query graph for global optimization.
2. Search the query graph in the post-order (i.e., search the graph recursively in the order of left child, right child, and parent) and identify basic patterns in it.
3. Optimize all type-NESTED_FORALL in a bottom-up fashion.
4. Evaluate the optimized query in a top-down fashion. For each level, apply appropriate optimization techniques.

□

4. CONSTRUCTIVE PLANNING

Parallel to queries, in an object base we define an update query to be a formula of the form:

declarations
action
such that conditions

Unlike queries, the purpose of update queries is to move an object base from one state to another state, subject to the constraints imposed by the system. The major problem with conjunctive update queries is that the operations in a query may be specified in an order such that the standard PROLOG evaluation process may fail to achieve the desirable purpose due to the so-called "negative goal interactions" [Wile83]. For instances, performing one operation may accidentally undo some previously accomplished operations, or performing one operation may prevent some other operations from being performed. The properly execute the operations, planning is required.

In the past, we have seen the major problem with linear planners: goal interactions. In many situations, goal interactions cannot be removed by simply reordering the operator sequence by which these goals are achieved. Rather, it requires that the operations be intermixed. Some nonlinear planners have been proposed based on this observation. Although nonlinear planners have been proven to be more efficient than linear ones, however, it can be proved that they have by no means solved general planning problems. Since robot task planning is a special class of planning problems, in this section we will show that some subclasses of robot task planning problems can be solved effectively.

4.1 NONLINEAR PLANNING AS CONSTRAINT SATISFACTION

In the simplest case, a nonlinear planner can first develop a plan for each conjunct goal, assuming that there is no interactions among these goals. Once the plans are developed, they are merged together based on the interactions among them. Initially, each plan provides a partial ordering among its own operations. As more interactions are uncovered, it may be necessary to move an operator from one plan to another place (so that certain interaction can be avoided) and consequently another partial ordering constraint is developed. For instance, if operator p in plan x negates the precondition of operator q of plan y, and x, y are executed in sequence initially. In this case, this negation may be avoided by moving q so that q is executed before p. In other words, a partial ordering constraint $p > q$ is developed.

We shall regard this as an operator ordering problem, and an ordering criterion has been proposed by Chapman in his *Modal truth Criterion* [Chap87] as described in the following.

Modal Truth Criterion

"A proposition p is necessarily true in a situation s iff two conditions hold: there is

a situation t equal or necessarily previous to s in which p is necessarily asserted; and for every step C possibly before s and every proposition q possibly codesignating with p which C *denies*[1], there is a step W necessarily between C and s which asserts r[2], a proposition such that r and p codesignate whenever p and q codesignate. The criterion for possibly truth is exactly analogous, with all the modalities switched (read "necessary" for "possible" and vice versa)."

For the operator ordering problem, consequently, let p be an ordering of operators $(f_1,...,f_n)$:

$p = [(f^0,a^0,e^0), (f^1,a^1,e^1),..., (f^n,a^n,e^n)]$,

where e^i stands for the effects created by operator f^i and a^i stands for the preconditions that have to be held before f^i can be applied. For simplicity, we assume that each e^i is a preposition in the operator ordering problem and a^i is true. Assume $G = \{u_1,...,u_n\}$ is a set of conjunctive goals that have to be satisfied, and for each u_i, $1 \le i \le n$, there is one and only one f_j such that $e^j => u_i$. An operator ordering problem with the above assumptions will be called a *simple* operator ordering problem later. For each u_i, let

$V_i = \{f_j | e^j => u_i\} = \{v_i\}$
$U_i = \{f_j | e^j => \tilde{} u_i\} = \{u_{i1},...,u_{iw(i)}\}$

Let $q(f_i)$ be the position of f_i in p, we can obtain that p is a plan if and only if for each i, the following is true:

$[q(v_i) > q(u_{i1})] \wedge ... \wedge [q(v_i) > q(u_{iw})]$

Let y_i denote the above formula, let let d_{ij} be $[q(f_i) > q(f_j)]$, and let D be the set $\{d_{ij} |$ there exists a y_k such that $d_{ij} \in y_k\}$. The problem of searching for an ordering of $(f_1,...,f_n)$ is equivalent to the problem of instantiating $(q(f_1),..., q(f_n))$ such that $y_1 \wedge ... \wedge y_n$ is true. To determine if a simple operator ordering problem is solvable, we can first enlarge D to D' with the transitive rule:

$d_{ij} \wedge d_{jk} \rightarrow d_{ik}$, for all possible i, j, and k.

Now, the problem is solvable if there exist no i and j such that both d_{ij} and d_{ji} belong to D'.

If an operator ordering problem is determined to be solvable, assuming the cardinality of D' is r, a solution can be obtained according to Algorithm 4.1 as follows:

Algorithm 4.1

step 1

sequence = null;

---

1 C is called a *clobber* by Chapman.

2 W is called a *white knight*; the process which re-asserts r is called *declobbering*.

step 2

1. For i = 1 to n do

   If there exists no j such that d_{ij} belongs to D', append f_i to the end of *sequence*;

2. For all i identified above, remove each d_{ki} for any k from D'. If no such i could be identified in (1) terminate; otherwise go to step 2.

□

According to step 2, the complexity of this algorithm is $O(r)$. The idea of assuming that the effects and preconditions of operators as static, state independent propositions and treating a conjunctive planning problem as a constraint satisfaction problem is not new. Such an idea has been explored in DCOMP [Nils80]. In DCOMP, a goal reduction process first develops an AND tree by expanding top-level goals into more detailed ones, assuming that all the goals are independent. However, the goal reduction process provides a partial ordering among higher-level goals and lower-level goals. An analysis of possible goal interactions is then carried out. As a consequence, for each goal, two lists are constructed. The first list, called the *add list*, contains all the subgoals whose effect can produce a precondition of the associated goal. The second list, called the *delete list*, contains all the subgoals whose effect can negate a precondition of the associated goal. Based on the add and delete lists, further ordering constraints can be developed, and sometimes new steps are added to satisfy the ordering constraints.

Like STRIPS, a hierarchical version of DCOMP, called NOAH, was proposed [Sace75]. Several researchers [AlKo83] [McDe83] [MFDD85] [Vere83] extended NOAH by improving the representation of time in different ways. David Wilkin's SIPE [Wilk84] further extended it with the concept of *resource*, by which operations utilizing shared resources can be sequenced to avoid conflicts, and with the mechanism that can perform simple deductions based on the effects produced by operators. More recently, based on the Modal Truth Criterion, David Chapman's TWEAK [Chap87] proposed a nonlinear approach that can be proved to be complete, i.e., the planner can develop a plan if it indeed exists. However it may never stop if such a plan does not exist, and Chapman showed that this is the best we can hope for.

TWEAK can be regarded as the first planning mechanism which is developed based on a sound theory. Like other nonlinear planners, it is constraint-based, so that a plan is a partial order of operators. Basically, given a set of goals, TWEAK tries to accomplish the goals by constructing and refining partial plans, where a partial plan is a set of steps for which some information (e.g., variables, orders) is not specified. Constraints are developed as the planning process proceeds. A typical constraint could be "*The variables x and y cannot codesignate (i.e., be unified).*"; or it could be a partial ordering constraint like "*The action x has to be executed before action y, although they do not have to be performed back to back.*" At any time, the planner has to make sure that the Modal Truth Criterion is satisfied, i.e., the preconditions of an action are always true, regardless how the current partial plan is completed. Assuming p is a precondition for an action that has been included in a partial plan, and s is the current situation, TWEAK employs one of the following techniques for this purpose:

1. *Simple Establishment*: If p has been established in s, the planner constrains a step that establishes p to occur before s.

2. *Step Addition*: A step is added right before s in order to assert p.
3. *Promotion*: A clobber is moved backward so that it will happen after s.
4. *Separation*: If q is a clobbering proposition which possibly codesignates with p, the planner constrains the partial plan by not allowing q to codesignate with p.
5. *Declobbering by White Knight*: Insert a white knight to avoid clobbering

We can discover that many of the above techniques have been employed in some previous planners, for which such techniques were proposed as heuristics. However, Chapman can prove that the above techniques are necessary and sufficient for constructing a complete and correct planner, and this is the major contribution of TWEAK.

Based on the above, a TWEAK planner can work in a straightforward way. Initially, given a problem, the plan is empty. The planner always looks for an unaccomplished goal to satisfy, and partial ordering constraints are incrementally introduced based on the Modal Truth Criterion as the planning process proceeds. One of the above techniques (e.g., goal promotion, white knights) is used to resolve a possible goal interaction whenever it occurs, and possibly new operators are introduced. If several alternatives exist to order an operator, to choose an operator, or to choose a goal, a nondeterministic choice is made. Whenever a new constraint is inconsistent with the existing ones, backtracking ia advocated based on dependency. Chapman showed that virtually almost all existing nonlinear planners could be regarded as a special case of TWEAK; and indeed he has cleaned up much of the research on general-purpose planning in the last decade. However, it should be noted that TWEAK has by no means solved the general planning problem [Amst87]. For instances, it may not develop an optimal plan, and the restricted form of propositions on which the Modal Truth Criterion is based makes it impossible to represent nontrivial domains. A more detailed discussion about its weaknesses can be found in [Chap87] [Amst87].

Example 4-1

Assume that the initial configuration and the goal configuration of a blocks world problem are given and shown in Figure 4.1(a) and Figure 4.1(b), respectively. Also assume that the available robot operations are *puton* and *putdown* (see Figure 4.2), where x stands for post-conditions and p stands for preconditions. Figure 4.3 shows how a constraint-based planner can solve this problem by incrementally establishing a plan. Note that in Figure 4.3(a), an arc between two operators $a \rightarrow b$ designates a precedence relationship, which is supposed to be derived from a constraint discovered in Figure 4.3(b).

□

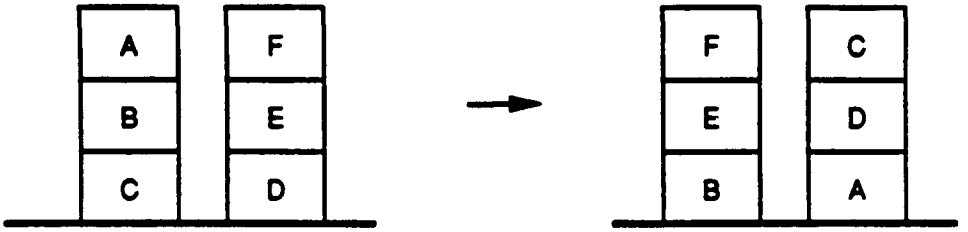
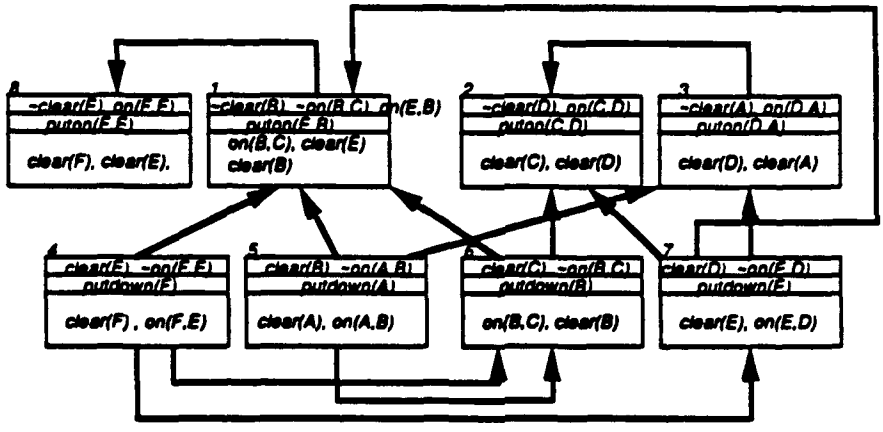


Figure 4.1 A blocks world problem

puton(x,y):
P: clear(x), clear(y)
X: (if on(x,z) then ~on(x,z)), on(x,y)
putdown(x):
P: clear(x)
X: if on(x,z) then ~on(x,z)

Figure 4.2 Robot operations



Goal	Operator	New Goal	Constraint	Reason
on(E,B)	puton(E,B)	clear(E), clear(B)	None	
clear(E)	putdown(F)	on(F,E)	4 < 1	4 produces precond. for 1
clear(B)	putdown(A)	None	5 < 1	5 produces precond. for 1
on(C,D)	puton(C,D)	clear(C), clear(D)	None	
clear(C)	putdown(B)	None	6 < 2	6 produces precond. for 2
			5 < 6	5 produces precond. for 6
			6 < 1	1 negates precond. of 6
clear(D)	putdown(E)	None	7 < 3	7 produces precond. for 3
			4 < 7	4 produces precond. for 7
			7 < 2	7 produces precond. of 2
on(D,A)	puton(D,A)	None	4 < 6	6 negates precond. of 4
			3 < 2	2 negates precond. of 3
			5 < 3	3 negates precond. of 5
on(F,E)	puton(F,E)	None	1 < 8	8 negates precond. of 1
None				
A Valid Total Ordering:			4, 7, 5, 6, 1, 3, 2, 8	

Figure 4.3 Execution of a non-linear planner

Example 4-2

This example is another illustration of a constraint-based planning process, assuming that the initial configuration and the goal configuration of a blocks world are given and shown in Figure 4.4(a) and Figure 4.4(b), respectively. Also assume that the available robot operations are the same as those in Example 4-1. Figure 4.5 shows the flow of the process. Note that in this case it is not able to derive an optimal plan, as blocks F and E need not be putdown before being stacked on A and F, respectively.

□

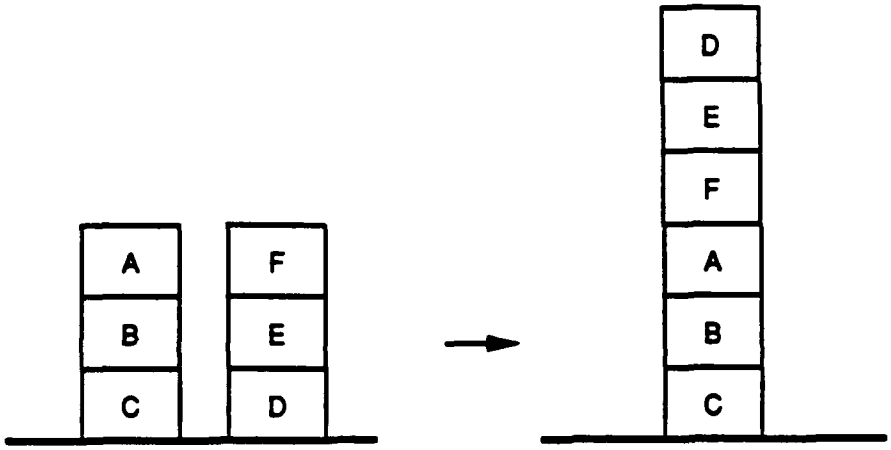
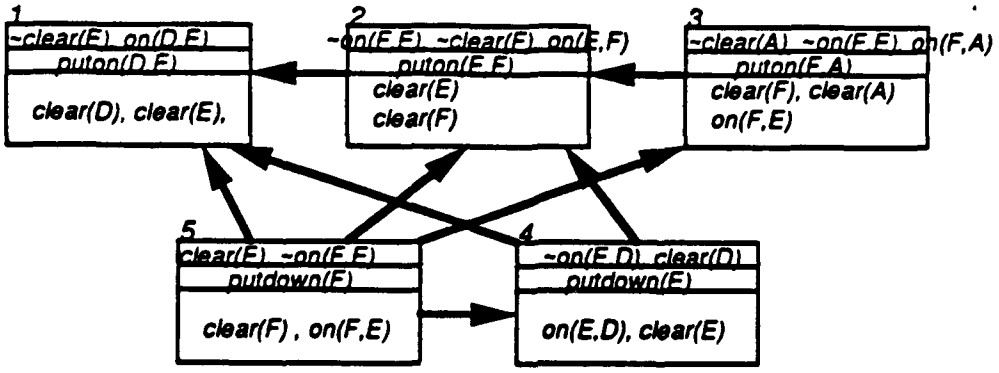


Figure 4.4 A blocks world problem



Goal	Operator	New Goal	Constraint	Reason
on(D,E)	puton(D,E)	clear(E), clear(D)	None	
clear(E)	putdown(F)	None	5 < 1	5 produces precond. for 1
clear(D)	putdown(E)	None	4 < 1	4 produces precond. for 1
			5 < 4	5 produces precond. for 4
on(E,F)	puton(E,F)	clear(E), clear(F)	2 < 1	1 negates precond. of 2
			5 < 2	5 produces precond. of 2
			4 < 2	4 may negate effect of 2.
on(F,A)	puton(F,A)	None	3 < 2	2 may negate effect of 3
			5 < 3	5 negates effect of 3

A Valid Total Ordering: 5, 4, 3, 2, 1

Figure 4.5 Execution of a non-linear planner

4.2 THE CONSUMER ORDERING PROBLEM

In some cases, the operators in an operator ordering problem work under the same set of resources and each of them would consume a (different) subset of the resources in order to accomplish its corresponding goal. On the other hand, the applicability of each operator is determined by the availability of certain resources. We call such a problem as the a *consumer ordering problem.*

Given a set of goals $(u_1,....,u_n)$ and a set of corresponding operators $(f_1,....,f_n)$, a consumer ordering problem can be formulated as follows. Let R be a set of resources and let R_i be the set of resources that will be consumed by operator f_i after it is applied. At each state s_j (which is defined to be the state of the system after the first j operators have been applied), in additional to the requirement that R_i has to be available, let $p_i(s_j)$ be the function that determines the applicability of f_i (i.e., it is true if f_i can be applied at state s_j, and it is false otherwise). Assuming the initial state is s_0, the corresponding consumer ordering problem is to find an ordering p of $f_1,....,f_n$ such that if f_i is the $j+1$th operator to be applied, $p_j(s_i)$ is true and R_i is available.

Clearly, the approach to solve a consumer ordering problem should be different from that of a simple operator ordering problem. This is because the preconditions of each operator are state dependent (except that the resources to be consumed are known), and therefore we cannot analyze the operators for any potential goal conflicts. However, we could have the observation that when the last operator is applied, the state of the system is completely predictable (i.e., the resources consumed by the other operators are known). This suggests that we can identify the last operator by choosing among the operators the one which is applicable, assuming all the other operators have been applied. If more than one of such operator can be identified, then they can be applied in an arbitrary order. Subsequently, we can remove these operators from further consideration, and this process is repeated until all the operators are identified. We shall call this procedure a *backward operator search process* later. Formally, this process can be described as follows, assuming the notations defined earlier are used:

Algorithm 4.2

step 1

sequence = null;
remained $= \{f_1,....,f_n\}$
$R' = R$;

step 2

1. $Q = \phi$;
   For each f_i in *remained* do

   Let $D = R' - R_i$ and let s be the state at which the resources in D have been occupied;

   If the intersection of $(R_1 \cup ... \cup R_{i-1} \cup R_{i+1} \cup ... \cup R_n)$ and R_i is not empty, there is a directly resource conflict so terminate; no solution exists for the problem.

   If $p_i(s)$ is true, append f_i to the end of *sequence*, let $R' = R' - R_i$, and let $Q = Q \cup \{f_i\}$;

2. For each f_i in Q, remove f_i from *remained*; if *remained* is ϕ terminate, else go to step 1.

□

4.3 THE PRODUCER ORDERING PROBLEM

A *producer ordering problem* is like a consumer ordering problem, except that instead of consuming some resources, each operator would produce some resources. However, the applicability of each operator is as well determined by a function, which is state dependent.

Given a set of goals $(u_1,...,u_n)$ and a set of corresponding operators $(f_1,...f_n)$, a producer ordering problem can be formulated as follows. Let R be a set of resources and let R_i be the set of resources that will be produced by operator f_i after it is applied. At each state s_j (which is defined to be the state of the system after the first j operators have been applied), let $p_i(s_j)$ be the function that determines the applicability of f_i (i.e., it is true if f_i can be applied at state s_j, and it is false otherwise). Assuming the initial state is s_0, the corresponding producer ordering problem is to find an ordering p of $f_1,...f_n$ such that if f_i is the j+1th operator to be applied, $p_j(s_i)$ is true.

A producer ordering problem as described above can be solved by a *forward search* process. Starting from the initial state, we first identify a set of operators that can be applied. Since each operator would simply produce some resources, the order of these operators would be irrelevant. The above process can be repeated, assuming all the operators just identified have been applied. This process continues until no more operators remained to be identified. Formally, this process can be described as follows, assuming the notations defined earlier are used:

Algorithm 4.3

step 1

sequence = *null;*
remained = $\{f_1,...f_n\}$
R = the set of resources that are available initially;

step 2

1. $Q = \phi$;
2. For each f_i in *remained* do

   Let $D = R \cup \{R_i \mid f_i \in Q\}$ and let s be the state at which all the resources in D are available.

   If $p_i(s)$ is true, append f_i to the end of *sequence* and let $Q = Q \cup \{f_i\}$; If *remained* is not empty and no such f_i exists, there is no solution to the problem and terminate.
3. For each f_i in Q, remove f_i from *remained*; if *remained* is ϕ terminate, else go to step 2.

□

4.4 THE CONSUMER-PRODUCER ORDERING PROBLEM

In addition to consumer ordering problems and producer ordering problems, there are situations in which the resources need to be reorganized, which would need both consumers and producers. We call such problems as *consumer-producer ordering problems.*

Given a set of goals $(u_1,...,u_n)$, a set of producer operators $(g_1,...,g_m)$, and a set of consumer operators $(f_1,...,f_n)$, a consumer-producer ordering problem can be formulated as follows. Let R be a set of resources, let G_i be the set of resources that will be produced by operator g_i after it is applied, and let R_i be the set of resources that will be consumed by operator f_i after it is applied. At each state s_j (which is defined to be the state of the system after the first j operators have been applied), in additional to the requirement that R_i has to be available, let $p_i(s_j)$ be the function that determines the applicability of f_i (i.e., it is true if f_i can be applied at state s_j, and it is false otherwise). Similarly, at each state s_j, in additional to the requirement that R_i has to be available, let $q_i(s_j)$ be the function that determines the applicability of g_i (i.e., it is true if g_i can be applied at state s_j, and it is false otherwise). Assuming the initial state is s_0, we can formulate the corresponding consumer-producer ordering problem as to find an ordering p of $f_1,...,f_n$, $g_1,...,g_m$ such that if f_i is the j+1th operator to be applied, $p_j(s_i)$ is true and R_i is available. Furthermore, if g_i is the j+1th operator to be applied, $q_j(s_i)$ is true and G_i is available.

Clearly, a consumer-producer ordering problem as formulated above can be solved easily by a two-phase approach. In the first phase, we sequence the producer operations according to Algorithm 4.3. In the second phase, the consumer operations are sequenced according to Algorithm 4.2. The consumer-producer ordering problem can be made more interesting by creating the concept of "consumer-producer-pair". First, we shall assume that the number of consumer operations and the number of producer operations are the same. Second, for each g_i, $1 \le i \le n$, an f_j, $1 \le j \le n$, is assigned to be its partner. Now, the consumer-producer ordering problem is to sequence f_1, ..., f_n, g_1, ..., g_n into a plan for which the number of consumer-producer-pairs is maximal, where we define a consumer-producer-pair of a plan to be any (f_i,g_j) pair, where g_j is the producer partner of f_i, that can be executed consecutively. It should be noted that any plan generated by the two-phase approach described earlier would have at most one consumer-producer-pair.

A consumer-producer ordering problem as formulated above can be solved in the following way. First, identify a partial ordering among the producer operations according to Algorithm 4.5 (see below). Second, identify a partial ordering among the consumer operations according to Algorithm 4.4 (see below), assuming that all producer operations have been applied.

Algorithm 4.4 (Partial Ordering Consumers)

step 1

sequence = null;
remained = $\{f_1,...,f_n\}$
index = 0;
$R' = R$;

step 2

1. $index = index + 1$; $T_{index} = \phi$;
2. For each f_i in *remained* do

   Let $D = R - R_i$ and let s be the state at which the resources in D have been occupied;

   If the intersection of $(R_1 \cup ... \cup R_{i-1} \cup R_{i+1} \cup ... \cup R_n)$ and R_i is not empty, there is a directly resource conflict so terminate; no solution exists for the problem.

   If $p_i(s)$ is true, $T_{index} = T_{index} \cup \{f_i\}$ and $R' = R' - R_i$;
3. For each f_i in T_{index}, remove f_i from *remained*; if *remained* is ϕ go to step 4; else go to step 1.
4. For $i = index-1$ to 1 *do*

   For each f_j in T_{i+1} *do*

   For each f_k in T_i *do*

   $f_j < f_k$;

□

Algorithm 4.5 (Partial Ordering Producers)

<u>*step 1*</u>

sequence = null;
remained = $\{f_1,...,f_n\}$
index = 0;
R = the set of resources that are available initially;
$Q = \phi$;

<u>*step 2*</u>

1. $T_{index} = \phi$;
2. For each f_i in *remained* do

   Let $D = R \cup \{R_i \mid f_i \in Q\}$ and let s be the state at which all the resources in D are available.

   If $p_i(s)$ is true, $T_{index} = T_{index} \cup \{f_i\}$; $Q = Q \cup \{f_i\}$; If *remained* is not empty and no such f_i exits, there is no solution to the problem and terminate.
2. For each f_i in T_{index}, remove f_i from *remained*; if *remained* is ϕ go to step 3, else *index* = *index* + 1 and go to step 1.
3. For $i = 1$ to *index*-1 *do*

   For each f_j in T_i *do*

   For each f_k in T_{i+1} *do*

   $f_j < f_k$;

□

The partial ordering developed in Algorithm 4.4 designates the minimal (logical) sequence requirement for the consumers. Similarly, the partial ordering developed in Algorithm 4.5 designates the minimal (logical) sequence requirement for the producers. Based on these basic requirements, the following algorithm develops a plan that maximizes the number of consumer-producer-pairs. This algorithm develops a plan incrementally. At each step, assuming the state is s, it looks for a consumer-producer-pair (f_i,g_j) to apply, where (1) both f_i and g_j have no precedents according to the basic ordering requirements, (2) g_i is executable under s, (3) f_i is executable under $s \bullet g_j$, and (4) f_i does not block any producer in the future. If such an (f_i,g_j) cannot be found, it chooses any applicable producer operator and applies it. This process repeats until no producers remained to be ordered; at this point all the remaining consumer operations are applied following the basic ordering requirements.

Algorithm 4.6

step 1

If there remains no g_j to be ordered, append each remaining f_i to *plan*.

step 2

Compute $I = \{(f_i,g_j) | (f_i,g_j)$ is a consumer-producer-pair, there exists no f_k such that $f_k < f_i$, and there exists no g_r such that $g_r < g_j\}$. If I is ϕ, choose any g_j, for which there exists no g_k so that $g_k < g_j$, and append g_j to *plan*.

step 3

Choose any (f_i,g_j) in I for which (1) g_j is executable under s, (2) f_i is executable under $s \bullet g_j$, and (3) f_i does not block any producer in the in the future. If such (f_i,g_j) can be found, do the following:

a. Append (f_i,g_j) to *plan*;

b. $s = s \bullet g_j$;

c. $s = s \bullet f_i$;

d. Remove any partial ordering requirement in which f_i takes the precedence.

e. Remove any partial ordering requirement in which g_j takes the precedence.

f. go to step 2.

If no such pair can be found, choose any g_j, for which there exists no g_k so that $g_k < g_j$, and append g_j to *plan*. Also remove any partial ordering requirement involving g_j and go to step 2.

□

As a final remark, the consumer-producer ordering algorithm discussed has a feature that has been implemented in some other planning systems: incremental planning, such as the approach proposed by Waldinger [Wald77]. In Waldinger's approach, each subgoal is solved at one time, followed by a goal-violation checking. Whenever a proposed operator creates a protected goal violation, it is inserted

at an earlier point in the partial plan. A check of goal violation is then conducted again and, if another goal violation occurs, it is inserted at another, yet earlier, point in the plan. However, Wadinger' approach is not guided by any information, and therefore it could be inefficient.

4.5 SOLVING BLOCK WORLDS AS C-P PROBLEMS

Clearly, a block worlds problem can be regarded as a consumer-producer problem. To see this, we shall assume that the following operators are available:

1. *remove*(*block_id*,*position*)
2. *put*(*block_id*,*position*)

Note that these two types of operators are sufficient to solve a blocks world problem. To convert a blocks world problem into a consumer-producer problem, the *remove* operators can be regarded as producer operations, and the *put* operators can be regarded as consumer operations. Given the initial configuration and the final configuration of a blocks world, the problem is to remove all the blocks in the initial configuration and put the blocks to their desirable positions. Assuming there are n blocks, it is clear that both the number of producer operations and the number of consumer operations are n. For each block, the corresponding *put* and *remove* operators form a consumer-producer pair. Furthermore, the functions p_i and q_i, $1 \leq i \leq n$, as described earlier, can be determined based on the following physical constraints:

1. No block can be moved to a position that is not supported.
2. No block can be removed from a position that supports another object.

A blocks world problem can then be solved by Algorithm 4.6.

Example 4-3

Assume that the initial configuration and the goal configuration of a blocks world are given as in Figure 4.6. Formulating this problem as a consumer-producer problem, only two types of operations are needed: *remove* and *put* (see Figure 4.7). Figure 4.8 shows the flow of Algorithm 4.6.

□

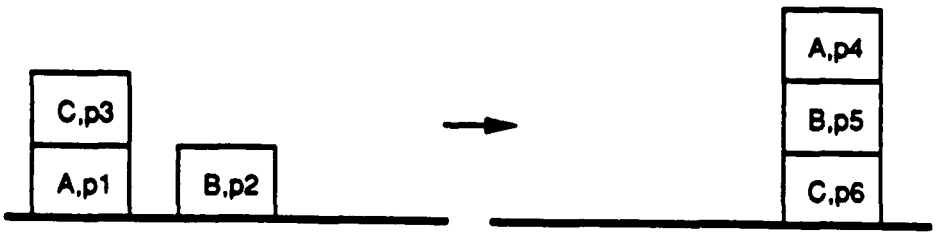


Figure 4.6 A blocks world problem

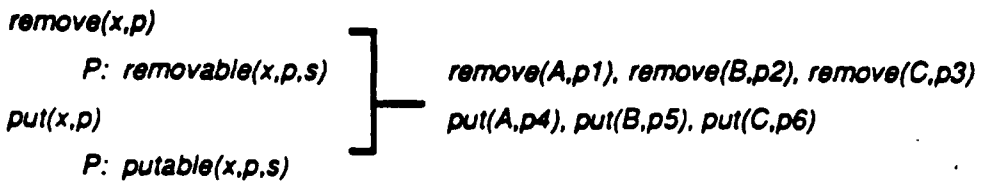


Figure 4.7 Robot operations

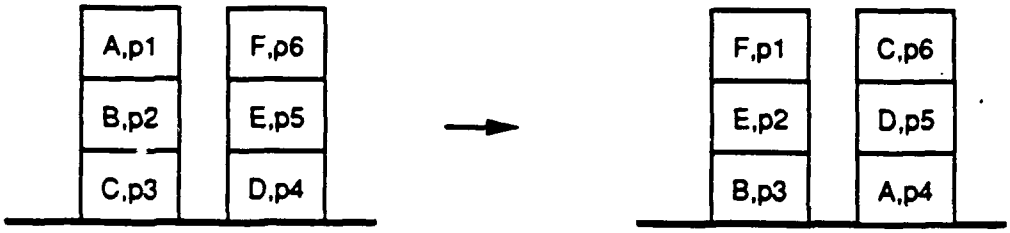
Consumers: 1: put(A,p4), 2: put(B,p5), 3:put(C,p6)
Producers: 4: remove(A,p1), 5: remove(B,p2), 6: remove(C,p3)
Initial Partial Ordering: 6 < 4, 3 < 2, 2 < 1
Consumer-Producer-Pair Found in Ist Iteration: (6,3)
Partial Ordering Constraints after 1st Iteration: 3 < 2, 2 < 1
Consumer-Producer-Pair Found in 2nd Iteration: (5,2)
Partial Ordering Constraints after 2nd Iteration: None
Consumer-Producer-Pair Found in 3rd Iteration: (4,1)

Figure 4.8 Execution of Algorithm 4.6

Example 4-4

Assume that the initial configuration and the goal configuration of a blocks world are given as shown in Figure 4.9. Formulating this problem as a consumer-producer problem, the possible operators are shown in Figure 4.10. Figure 4.11 shows the flow of Algorithm 4.6.

□



(a) initial configuration (b) goal configuration

Figure 4.9 A blocks world problem

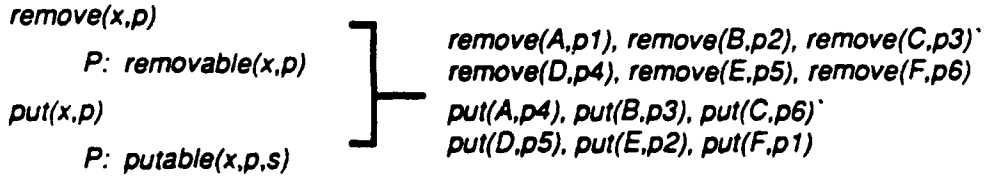


Figure 4.10 Robot operations

Consumers: 1: put(A,p4), 2: put(B,p3), 3:put(C,p6),
4: put(D,p5), 5:put(E,p2), 6:put(F,p1)
Producers: 7: remove(A,p1), 8: remove(B,p2), 9: remove(C,p3)
10:remove(D,p4), 11:remove(E,p5), 12:remove(F,p6)
Initial Partial Ordering: 7 < 8 < 9, 12 < 11 < 10, 2 < 5 < 6, 1 < 4 <3
Consumer-Producer-Pair Found in !st Iteration: None
Producer Applied in 1st Iteration: 12
Partial Ordering Constraints after 1st Iteration: 7 < 8 < 9, 2 < 5 < 6, 11 < 10, 1 < 4 < 3
Consumer-Producer-Pair Found in 2nd Iteration: None
Producer Applied in 2nd Iteration: 7
Partial Ordering Constraints after 2nd Iteration: 8 < 9, 11 < 10, 2 < 5 <6, 1 < 4 < 3
Consumer-Producer Pair Found in 3rd Iteration: None
Producer Applied in 3rd Iteration: 8
Partial Ordering Constraints after 3rd Iteration: 11 < 10, 2 < 5 < 6, 1 < 4 < 3
Consumer-Producer Pair Found in 4th Iteration: None
Producer Applied in 4th Iteration: 11
Partial Ordering Constraints after 4th Iteration: 2 < 5 < 6, 1 < 4 < 3
(All producers have been applied at this point)
Consumers Applied Finally: 2, 1, 5, 4, 3, 6

Figure 4.11 Execution of Algorithm 4.6

Example 4-5

Assume that the initial configuration and the goal configuration of a blocks world are given as shown in Figure 4.12. Formulating this problem as a consumer-producer problem, the possible operators are shown in Figure 4.13. Figure 4.14 shows the flow of Algorithm 4.6. Note that in this case Algorithm 4.6 is able to develop an optimal plan.

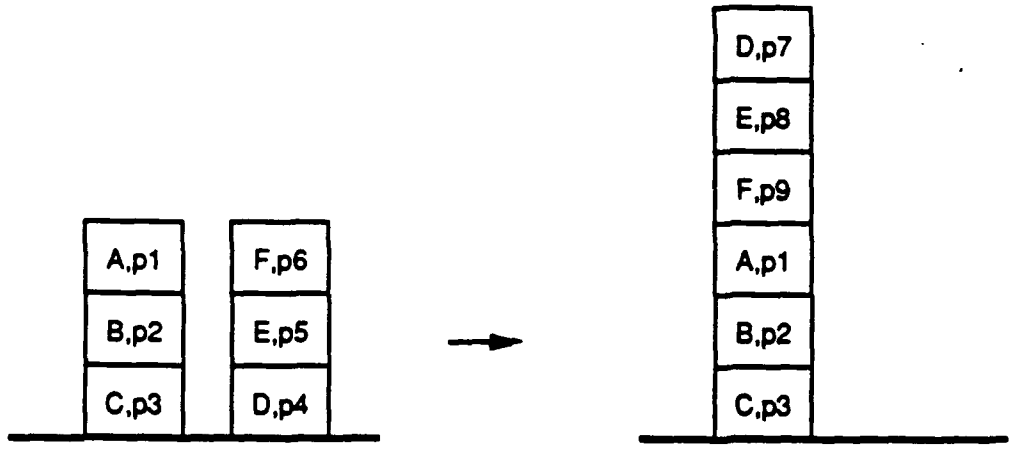


Figure 4.12 A blocks world problem

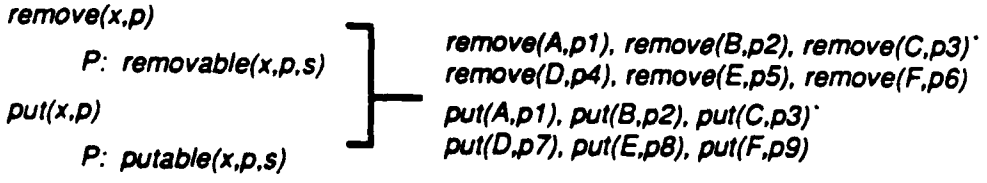


Figure 4.13 Robot operations

Consumers: 1: put(D,p7), 2:put(E,p)8, 3:put(F,p9)
Produc ers: 4:remove(D,p4), 5::remove(E,p5), 6:remove(F,p6)
Initial Partial Ordering: 6 < 5 < 4, 3 < 2 < 1
Consumer-Producer-Pair Found in !st Iteration: (6,3)
Producer Applied in 1st Iteration: 6
Partial Ordering Constraints after 1st Iteration: 5 < 4, 2 < 1
Consumer-Producer-Pair Found in 2nd Iteration: (5,2)
Producer Applied in 2nd Iteration: 6
Consumer-Producer Pair Found in 3rd Iteration: (4,1)
Producer Applied in 3rd Iteration: 4
Partial Ordering Constraints after 3rd Iteration: None
(All producers and Consumers have been applied at this point)

Figure 4.14 Execution of Algorithm 4.6

5. REFERENCES

[AlKo83] Allen, J.F., and Kooman, J.A., "Planning Using a Temporal World Model," *Proceedings, IJCAI*, 1983, pp. 741-747.

[Amst87] Amsterdam, J., "Planning with TWEAK," *AI Expert*, January, 1987, pp. 28-32.

[Astr76] Astrahan, M. et. al., "System R: Relational approach to database management," *ACM Trans. Database Syst.*, Vol. 1, No. 2, June 1976, pp 97-137.

AtBu87 Atkinson, M., and Buneman, O., "Types and Persistence in Database programming Languages," *ACM Computing Surveys*, Vol. 19, No.2, June, 1987, pp. 105-190.

[BABK87] Burton, B., Aragon, R.W., Bailey, S, Koehler, K.D., and Mayes, L.A., "The Reusable Software Library," *IEEE Software*, July, 1987, pp. 25-33.

[Bars87] Barstow, D., "Artificial Intelligence and Software Engineering," *9th International Conference on Software Engineering*, 1987, pp. 200-211.

[BGGH91] Blair, Gallagher, Hutchison, and Shepherd (eds.), *Object-Oriented Languages, Systems and Applications*, John Wiley & Sons, 1991.

[BoMu84] Boyle, J., and Muralidharan, M., "Program Reusability through Program Transformation," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, Sept. 1984, pp. 574-588.

[BMPP89] Bauer, F., Moller, B., Partsch, H., and Pepper, P., "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming," *IEEE Trans. on Software Engineering*, Vol. 15, No. 2, Feb. 1989, pp. 165-180.

[CACM91] *Communications of The ACM*, Special issue on object-oriented database systems, Vol. 34, 10, October, 1991.

[Carb81] Carbonell, J., "A Computational Model of Analogical Problem Solving," *Proceedings of IJCAI*, 1981, pp. 147-152.

[Care86] Carey, M., DeWitt, D., Richardson, J., and Sheikta, E., "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the 12th International Conference on VLDB*, Kyoto, Japan, Aug. 1986.

[Chap87] Chapman, D., "Planning for Conjunctive Goals," *Artificial Intelligence*, Vol. 32, 1987, pp. 333-377.

[Chea84] Cheatham, T., " Program Reusability Through Program Transformation," *IEEE Trans. Software Engineering, SE-10.5*, Sept. 1984.

[ChLe69] Cnang, C.L., and Lee, R.C.T., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1969.

[ChMi82] Chakravarthy, U., and Minker, J., "Processing Multiple Queries in Database Systems," *Database Engineering*, Vol. 1, 1982.

[ChMi86] Chakravarthy, U., and Minker, J., "Multiple Query Processing in Deductive Databases using Query Graphs," *Proceedings of the 12th International Conference on VLDB*, Kyoto, August 1986, pp 384-391.

[CIP84] CIP Language Group, *Lecture Notes in Computer Science. Volume I: The Munich project CIP*, Spring-Verlag, 1984.

[Clem88] Clemm, G., "The Workshop System - A Practical Knowledge-Based Software Environment," *SIGSOFT '88 Third Symp. on Software Development Environments*, Boston, MA., Nov. 1988, pp 55-64.

[CoMa84] Copeland, G., and Maier, D., "Making Smalltalk a database system," *Proceedings of SIGMOD*, ACM, New York, 1984, pp 316-325.

[Ders86] Dershowitz, N., "Programming by Analogy," in *Machine Learning: An Artificial Intelligence Approach, Vol. 2*, Michalski, R., Carbonell, J., and Mitchell, T. eds, Morgan Kaufmann Publishers, Inc., 1986, pp. 395-423.

[Dill88] Dillistone, B., "Configuration Management within an IPSE and its Implications for Software Re-use," in *Software Engineering Environments*, Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[Estu86] Estublier, B., "Experience with A Data Base of Programs," *Second Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, Dec. 1986, pp. 84-91.

[Feat86] Feather, M., "A Survey and Classification of some Program Transformation Approaches and Techniques," *Working Conference on Program Specification and Transformation*, Tolz, FRG, April 1986.

[Fish87] Fish, D. et al., "Iris: An Object-Oriented Database management System," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, pp48-69.

[GoAR89] Goldberg, Adele, and Roboson, *Smalltalk-80, The Language*, Addison-Wesley, 1989.

[GoWo87] Ganski, R., and Wong, H., "Optimization of Nested SQL Queries Revisited," *Proc. ACM SIGMOD*, May 1987, pp 23-33.

[GrMi80] Grant, J., and Minker, J., "On Optimizing the Evaluation of a Set of Expressions," Technical Report TR-916, University of Maryland, College Park, MD, July 1980.

[GrMi81] Grant, J., and Minker, J., "Optimization in Deductive and Conventional Relational Database Systems," In *Advances in Database Theory, Vol. 1*, H. Gallaire, J. Minker and J. Nicolas, Eds., Plenum Press, New York, 1981.

[HuKi88] Hudson, S., and R. King, " The Cactis Project: Database Support for Software Environments," *IEEE Trans. on Software Engineering*, Vol. 14, No. 6, June 1988, pp 709-719.

[JaKo84] Jarke, M., and Koch J., "Query Optimization in Database Systems," *ACM Computing Survey*, Vol. 16, June 1984.

[Jark84] Jarke, M., "Common Subexpression Isolation in Multiple Query Optimization," In *Query Processing in Database Systems*, W. Kim, D.Reiner and D. Batory, Eds., Springer-Verlag, New York, 1984.

[KaGa87] Kaiser, G. E., and Garian, D., " Melding Software Systems from Reusable Building Blocks," *IEEE Software* , July 1987, pp. 17-24. .

[Kim82] Kim, W., "On optimizing an SQL-like nested query," ACM Trans. Database Syst., Vol. 7, No. 3, Sept. 1982, pp 443-469.

[Kim84] Kim, W., "Global Optimization of Relational Queries: A First Step," In *Query Processing in Database Systems*, W. Kim, D.Reiner and D. Batory, Eds., Springer-Verlag, New York, 1984.

[Koen92] Koenig, A., "Designing a C++ Container Class," *Journal of Object-Oriented Programming*, 4, 9, February, 1992, pp. 37-39.

[Kuck81] Kuck, D. et al, "Dependence Graphs and Compiler Optimizations," *Proc. of the 8th ACM Symp. on Principles of Programming Languages*, 1981, pp 207-218.

[MaSt86] Maier, D., and J. Stein, "Development of an Object-Oriented DBMS," *OOPSLA '86 Proceedings*, 1986, pp472-481.

[MaWa79] Manna, Z., and Waldinger, R., "Synthesis: Dreams → Programs," *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 4, July 1979, pp. 294-328.

[McDe83] McDermott, D., "Generalizing Problem Reduction: A Logical Analysis," *Proceedings, IJCAI*, 1983.

[MeFe81] Medina-Mora, R., and Feiler, P., "An Incremental Programming Environment," *IEEE Trans. on Software Engineering*, Vol. SE-7, Sept. 1981.

[MFDD85] Miller, D., Firby, R.J., Dean, T., "Deadlines, Travel Time, and Robot Problem Solving," *Proceedings, IJCAI*, 1985.

[MiDG86] Mili, A., Desharnais, J., and Gagne, J., "Formal Models of Stepwise Refinement of Programs," *ACM Computing Surveys*, Vol. 18, No. 3, September 1986, pp 231-276.

[Nils80] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[Obri86] O'brien, P., Bullis, B., and Schaffert, C., "Persistent and Shared Objects in Trellis/Owl," *Proc. 1986 IEEE Workshop on Object-oriented Database Systems.*

[PaSe88] Park, J., and Segev, A., "Using common subexpressions to optimize multiple queries," *4th International Conference on Data Engineering*, Feb., 1988

[PaSt83] Partsch, H., and Steinbruggen, R., "Program Transformation Systems," *ACM Computing Surveys*, Vol.15, No. 3, September 1983, pp 199-236.

[PaTL89] Park, J., Teorey, T., and Lafortune, S., "A Knowledge-based approach to multiple query processing," *Data and Knowledge Engineering*, Vol. 3, North-Holland, 1989.

[NTFT91] Nishida, F., Takamatsu, S., Fujita, Y., and Tani, T., "Semi-Automatic Program Construction From Specifications Using Library Modules," *IEEE Transactions on Software Engineering*, Vol.17, No. 9, September 1991, pp. 853-871.

[Pene86] Penedo, M., "Prototyping a Project Master Data Base for Software Engineering Environments," *Second Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, Dec. 1986, pp. 1-11.

[PrFr87] Prieto-Diaz, R., and Freeman, P., "Classifying Software for Reusability," *IEEE Software*, Vol. 4, No. 1, January 1987, pp 6-16.

[PrSm88] Pressburger, T., and Smith, D., "Knowledge-based Software Development Tools," in *Software Engineering Environments*, Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[RoSt87] Rowe, L., and M. Stonebraker, "The POSTGRES Data Model," *Proceedings of VLDB*, 1987, pp83-96.

[Sace75] Sacerdoti, E.D. "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence*, 1974, pp115-135 Note 109, AI Center, SRI, 1975.

[Scha90] Schach, S. R., *Software Engineering*, Aksen Associates, 1990.

[Schm77] Schmidt, J., "Some high level language constructs for data of type relation," *ACM Trans. on Database Systems*, Vol. 2, No. 3, September, 1977, pp 247-261.

[Sell88] Sellis, T., "Multiple-Query Optimization," *ACM Transaction on Database Systems*, Vol. 13, No. 1., March 1988. 1987, pp365-374.

[SmKW85] Smith, D., Kotik, G., and Westfold, S., "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Trans. On Software Engineering*, Vol. SE-11, No. 11, November 1985, pp 1278-1295.

[StHH87] Stonebraker, M., E. Hanson, and C. Hong, "The Design of the POSTGRES Rules System," *Proceedings of Data Engineering*, 1987, pp365-374.

[Stro91] Bjarne Stroustrup, *The C++ Programming Language*, 2nd Edition, Addison Wesley, 1991.

[SWKH76] Stonebraker, M., Wong, E., Kreps, P., and Held, G., "The design and implementation of INGRES," *ACM Trans. Database Syst.*, Vol. 1, No. 3, Sept. 1976, pp 189-222.

[TeRe81] Teitelbaum, T., and Reps, T., "The Cornell program synthesizer: A syntax-directed programming environment," *Communication of ACM*, Vol. 24, No. 9, Sept. 1981.

[Vere83] Vere, S.A., "Planning in Time: Windows and Durations for Activities and Goals", *IEEE Transaction on Machine intelligence and Pattern Analysis*, May, 1983, pp246-266.

[Wald77] Waldinger, R., "Achieving Several Goals Simultaneously", in *Elcock and Michie(Eds.), Machine Intelligence*, 8, pp94-136.

[Wate85] Waters, R., "The Programmer's Apprentice: A Session with KBSmacs," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 11, November 1985.

[Wilk84] Wilkins, D.E., "Domain-Independent Planning - Representation and Plan Generation", *Artificial Intelligence*, 22, 1984, pp269-301.

[WoBa87] Wolfe, M., and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming*, Vol. 16, No. 2, 1987, pp 137-178.

[WoSo88] Wood, M., and Sommerville, I., "A knowledge-based software components catalogue," in *Software Engineering Environments*, Pearl Brereton ed., Ellis Horwood Limited, England, 1988.

[WoYo76] Wong, E., and Youssefi, K., "Decomposition - A strategy for query processing," *ACM Trans. on Database Systems*, Vol. 1, No. 3, September, 1976, pp 223-241.